Universität Hamburg
Department Informatik
Vogt-Kölln-Str. 30
D-22527 Hamburg

Bericht 294

Proceedings of the International Workshop on Petri Nets and Software Engineering PNSE'10

FBI-HH-B-294/10

Herausgeber:
Michael Duvigneau
Daniel Moldt
Universität Hamburg
Department Informatik

In die Reihe der Berichte des Fachbereichs Informatik aufgenommen durch
Prof. Dr. M. Jantzen
Prof. Dr. W. Lamersdorf

Juni 2010

This document is retrievable

- on CD under the ISBN 978-972-8692-55-1
- as printed report (see address on front page)
- as electronical copy of the printed report under the URL
  http://epub.sub.uni-hamburg.de/informatik/volltexte/2010/148/

Abstract

This report contains the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'10) that took place in Braga, Portugal on June 22, 2010. For the successful realisation of complex systems of interacting and reactive software and hardware components the use of a precise language at different stages of the development process is of crucial importance. Petri nets provide a uniform language supporting the tasks of modelling, validation, and verification which captures fundamental aspects of causality, concurrency and choice in a natural and mathematically precise way without compromising readability.

The use of Petri nets (P/T-nets, coloured Petri nets and extensions) in the formal process of software engineering is presented as well as their application and tools supporting the disciplines mentioned above.

Zusammenfassung

Dieser Bericht enthält die Proceedings des internationalen Workshops über *Petrinetze und Softwaretechnik* (PNSE'10), der am 22. Juni 2010 in Braga, Portugal stattfand. Zur erfolgreichen Erstellung komplexer Systeme interagierender und reaktiver Software- und Hardwarekomponenten ist die Verwendung einer präzisen Sprache während der verschiedenen Entwicklungsstufen von größter Wichtigkeit. Petrinetze stellen eine einheitlich für die Zwecke der Modellierung, Validierung und Verifikation verwendbare Sprache dar, welche fundamentale Aspekte von Kausalität, Nebenläufigkeit und Alternativen auf eine natürliche und mathematisch präzise Weise einfängt, ohne die Lesbarkeit der Modelle zu beeinträchtigen.

Der Workshop präsentiert sowohl die Verwendung von Petrinetzen (S/T-Netzen, gefärbten Netzen und Erweiterungen) in den formalen Prozessen der Softwaretechnik als auch deren Anwendung oder Werkzeuge, welche die genannten Disziplinen unterstützen.

Editors: Michael Duvigneau and
Daniel Moldt

Proceedings of the International Workshop on

P etri
N ets and
S oftware
E ngineering
PNSE'10

University of Hamburg
Department of Informatics

Preface

This booklet contains the proceedings of the International Workshop on *Petru Nets and Software Engineering* (PNSE'10) in Braga, Portugal, June 22, 2010. It is a co-located event of *Petri Nets 2010*, the 31st international conference on Applications and Theory of Petri Nets and other Models of Concurrency, and *ACSD 2010*, the 10th International Conference on Application of Concurrency to System Design.

More information about the workshop, like online-proceedings, can be found at

`http://www.informatik.uni-hamburg.de/TGI/events/pnse10/`

For the successful realisation of complex systems of interacting and reactive software and hardware components the use of a precise language at different stages of the development process is of crucial importance. Petri nets are becoming increasingly popular in this area, as they provide a uniform language supporting the tasks of modelling, validation, and verification. Their popularity is due to the fact that Petri nets capture fundamental aspects of causality, concurrency and choice in a natural and mathematically precise way without compromising readability.

The use of Petri nets (P/T-nets, coloured Petri nets and extensions) in the formal process of software engineering, covering modelling, validation, and verification, is presented as well as their application and tools supporting the disciplines mentioned above.

The program committee consists of:

We received 16 high-quality contributions. The program committee has accepted four of them for full presentation. Furthermore the committee accepted

five papers as short presentations. Three contributions were submitted and accepted as posters.

The international program committee was supported by the valued work of Michal Knapik, Levi Lucio, Elisabetta Mangioni and Tarek Melliti as additional reviewers. Their work is highly appreciated.

Furthermore, we would like to thank the organizational teams of the Universidade do Minho and the Instituto Politécnico de Beja, Portugal, for their general organizational support.

Without the enormous efforts of authors, reviewers, PC members and the organizational teams this workshop wouldn't provide such an interesting booklet.

Thanks!

Michael Duvigneau and Daniel Moldt
Hamburg, June 2010

Contents

Part III Short Presentations

Part IV Poster Abstracts

Part I

Invited Talk

Combining Petri Nets and UML for Model-based Software Engineering

João M. Fernandes

Dep. Informática / CCTC
Universidade do Minho
4710-057 Braga
Portugal

EXTENDED ABSTRACT

UML is by far the most widely used modelling language used nowadays in software engineering, due to its large scope and its wide tool support. This software standard offers many diagrams that cover all typical perspectives for describing and modelling the software systems under consideration. Among those diagrams, UML includes diagrams (activity diagram, state machine diagram, use case diagrams, and the interaction diagrams) for describing the behaviour (or functionality) of a software system. Petri nets constitute a well-proven formal modelling language, suitable for describing the behaviour of systems with characteristics like concurrency, distribution, resource sharing, and synchronisation. Thus, one may question why not combining some UML diagrams with Petri nets for effectively supporting the activities of the software engineer. The usage of Petri nets for/in Software Engineering was addressed by several well-known researchers, like, for example, Reisig [6], Pezzè [1], Machado [5], and Kindler [4].

In this invited talk, we discuss some alternatives to introduce Petri nets into a UML-based software development process. In particular, we describe how Coloured Petri Net (CPN) models can be used to describe the set of scenarios associated with a given use case. We describe three different alternatives that can be adopted to achieve that purpose.

The first approach, initially presented in [7], suggests a set of rules that allow software engineers to transform the behaviour described by a UML 2.0 sequence diagram into a CPN model. Sequence diagrams in UML 2.0 are much richer than those in UML 1.x, namely by allowing several traces to be combined in a unique diagram, using high-level operators over interactions. The main purpose of the transformation is to allow the development team to construct animations based on the CPN model that can be shown to the users or the clients in order to reproduce the expected scenarios and thus validate them. Thus, non-technical stakeholders are able to discuss and validate the captured requirements. The usage of animation is an important topic in this context, since it permits the user to discuss the system behaviour using the problem domain language.

In the second approach, discussed in [3], we assume that developers specify the functionality of the system under consideration with use cases, each of which is described by a set of UML 2.0 sequence diagrams. For each use case, there

should exist at least one sequence diagram that represents and describes its main scenario. Other sequence diagrams for the same use case are considered to be variations of the main scenario. The transformation approach allows the development team to interactively play or reproduce any possible run of the given scenarios. In particular, the natural characteristics of the CPN modelling language facilitate the representation of the hierarchy and concurrency constructs of sequence diagrams.

The third alternative, considered in [2], is an improvement with respect to the previous approach and is targeted to reactive systems. We identify and justify two key properties that the CPN model must have, namely: (1) controller-and-environment-partitioned, which means constituting a description of both the controller and the environment, and distinguishing between these two domains and between desired and assumed behaviour; (2) use case-based, which means constructed on the basis of a given use case diagram and reproducing the behaviour described in accompanying scenario descriptions. We have demonstrated how this CPN model is useful for requirements engineering, since it provides a solid basis for addressing behavioural issues early in the development process, for example regarding concurrent execution of use cases and handling of failures.

References

1. G. Denaro and M. Pezzè. Petri Nets and Software Engineering. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 439–466. Springer, 2004. DOI 10.1007/b98282.
2. J.M. Fernandes, J.B. Jørgensen, and S. Tjell. Requirements engineering for reactive systems: Coloured petri nets for an elevator controller. In *14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, pages 294–301. IEEE CS Press, December 2007. DOI 10.1109/APSEC.2007.81.
3. J.M. Fernandes, S. Tjell, J.B. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *6th Int. Workshop on Scenarios and State Machines (SCESM 2007), within ICSE 2007*. IEEE CS Press, May 2007. DOI 10.1109/SCESM.2007.1.
4. Ekkart Kindler. Model-Based Software Engineering and Process-Aware Information Systems. *Transactions on Petri Nets and Other Models of Concurrency*, 5460:27–45, 2009. 10.1007/978-3-642-00899-3_2.
5. R. J. Machado, K. B. Lassen, S. Oliveira, M. Couto, and P. Pinto. Requirements Validation: Execution of UML Models with CPN Tools. *International Journal on Software Tools for Technology Transfer*, 9(3–4):353–369, 2007. DOI 10.1007/s10009-007-0035-0.
6. W. Reisig. Petri Nets in Software Engineering. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 63–96. Springer, 1987. DOI 10.1007/3-540-17906-2_22.
7. O. Ribeiro and João M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In K. Jensen, editor, *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 237–256, October 2006.

Part II

Long Presentations

The Resource Allocation Problem in Software Applications: A Petri Net Perspective*

Juan-Pablo López-Grao[1] and José-Manuel Colom[2]

[1] Dpt. of Computer Science and Systems Engineering (DIIS)
[2] Aragonese Engineering Research Institute (I3A)
University of Zaragoza, Spain
Email: {jpablo,jm}@unizar.es

Abstract. Resource Allocation Systems (RAS) have been intensively studied in the last years in the domain of Flexible Manufacturing Systems (FMS). The success of this research line has been based on the identification of particular subclasses of Petri Nets that correspond to a RAS abstraction of this kind of systems. In this paper we take a parallel road to that travelled through for FMS, but for the case of software applications. The considered applications present concurrency and deadlocks can happen due to the allocation of shared resources. We will evince that the existing subclasses of Petri Nets used to study this kind of deadlock problems are insufficient, even for very simple software systems. From this starting point we propose a new subclass of Petri Nets that generalizes the previously known RAS subclasses and we present a taxonomy of anomalies that can be found in the context of software systems.

1 Introduction

Among the most recurrent patterns in a wide disparity of engineering disciplines, the competition for shared resources between concurrent processes takes a prominent position. The reader might think of examples in the context of distributed systems, operations research, manufacturing plants, etc. The perspective of discrete event systems theory proves appropriate and powerful as a framework in which provide solutions to the so-called resource allocation problem [1]. Systems of this kind are often called Resource Allocation Systems (RAS) [2, 3].

RAS are usually conceptualized around two distinct entities, processes and resources, thanks to a prior abstraction process which is inherent in the discipline. The resource allocation problem refers to satisfying successfully the requests for resources made by the processes, ensuring that no process ever falls in a deadlock. A set of processes is deadlocked when they indefinitely wait for resources that are already held by other processes of the same set [4].

RAS can be categorized both on the type of processes (sequential, non-sequential) and resources (serially reusable, consumable) [5]. Hereafter, we will

* This work has been partially supported by the European Community's Seventh Framework Programme under Project DISC (Grant Agreement n. INFSO-ICT-224498) and the project CICYT-FEDER DPI2006-15390.

focus on Sequential RAS with serially reusable resources. This means that a process can increase or decrease the quantity of free resources during its execution. However, the process will contervail that operation before terminating, i.e. resources are used in a conservative way.

Although other models of concurrency have also been considered [6], Petri nets [7] have arguably taken a leading role among the family of formal models used for dealing with the resource allocation problem [8, 9]. One of the strengths of this approach is the smooth mapping between the main entities of RAS and the basic elements of Petri net models. A resource type can be modelled using a place: the number of instances of it being modelled with tokens. Meanwhile, sequential processes are modelled with tokens progressing through state machines. Arcs from resource places to transitions (from transitions to resource places) represent the acquisition (return) of some resources by a process. Petri nets thus provide a natural formal framework for the analysis of RAS, besides benefiting from the goods of compositionality.

This fact is well notorious in the domain of Flexible Manufacturing Systems (FMS), where Petri net models for RAS have widely succeeded since the seminal work of Ezpeleta et al. was introduced [8]. This is mostly due to a careful selection of the subclass of Petri nets used to model these FMS, based upon two solid pillars. First, the definition of a rich syntax from a physical point of view, which enables the natural expression of a wide disparity of plant configurations. And second, the contribution of sound scientific results which let us characterize deadlocks from the model structure, as well as provide a well-defined methodology to automatically correct them in the real system.

Nowadays, there exists a plethora of Petri net models for modelling RAS in the context of FMS, which often overcome some of the syntactical limitations of the S^3PR class [8]. S^4PR net models [10, 11] generalize the earlier, while allowing multiple simultaneous allocations of resources per process. S^*PR nets [12] extend the expressive power of the processes to that of state machines: hence internal cycles in their control flow is allowed. However, deadlocks in S^*PR net models are not fully comprehended from a structural perspective. Other classes such as NS-RAP [9], ERCN-merged nets [13] or PNR nets [14] extend the capabilities of S^3PR/S^4PR models beyond Sequential RAS by way of lot splitting or merging operations.

Most analysis and control techniques in the literature are based on the computation of a structural element which univocally characterizes deadlocks in many RAS models: the so-called *bad siphon*. A bad siphon is a siphon which is not the support of a p-semiflow. If bad siphons become (sufficiently) emptied, their output transitions die since the resource places of the siphon cannot regain tokens anymore, thus revealing the *deadly embrace*. Control techniques thus rely on the insertion of monitor places [15], i.e. controllers in the real system, which limit the leakage of tokens from the bad siphons.

Although there exist obvious resemblances between the resource allocation problem in FMS and that of parallel or concurrent software, previous attempts to bring these well-known RAS techniques into the field of software engineering

have been, to the best of our knowledge, either too limiting or unsuccessful. Gadara nets [16] constitute the most recent attempt, yet they fall in the over-restrictive side in the way the resources can be used, as a result of inheriting the design philosophy applied for FMS. In this work, we will analyze why the net classes and results introduced in the context of FMS fail when brought to the field of concurrent programming.

Section 2 presents a motivating example and discusses the elements that a RAS net model should desirably feature in order to successfully explore the resource allocation problem within the software enginering discipline. Taking into account those considerations, section 3 introduces a new Petri net class, called PC^2R. Section 4 relates the new class to those defined in previous works and establishes useful net transformations which forewarn us about new behavioural phenomena. Section 5 introduces some of these anomalies which highlight the fact that previous theoretical results in the context of FMS are insufficient in the new framework. Finally, section 6 summarizes the results of the paper.

2 The RAS view of a software application

Example 1 presents a humorous variation of Dijkstra's classic problem of the dining philosophers. We will adopt and adapt the beautiful writing by Hoare at [17] for its enunciation.

Example 1. The pragmatic dining philosophers. "Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. A microwave oven is also available. In the center of the table there is a large bowl of spaghetti which is frequently refilled (so it cannot be emptied), and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Then he touches the bowl to feel its temperature. If he feels the spaghetti got too cold, he will leave his fork and take the bowl to the microwave. Once it is warm enough, he will come back to the table, sit on his chair and leave the bowl on the table after recovering his left fork (please bear in mind that the philosopher is *really* hungry by now). Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. If he can do it before the bowl gets cold again, he will serve himself and start eating. When he has finished, he puts down both forks and leaves the room."

According to the classic RAS nomenclature, each philosopher is a sequential process, and the five forks plus the bowl are serially reusable resources which are shared among the five processes. From a software perspective, each philosopher can be a process or a thread which will be executed concurrently.

Algorithm 1 introduces the code for each philosopher. Notationally, we modelled the acquisition / release of resources by way of the `wait()` / `signal()` operations, respectively. Both of them have been generalized for the acquisition of multiple resources (separated by commas when invoking the function). Finally,

the `trywait()` operation is a non-blocking wait operation. If every resource is available at the time `trywait()` is invoked, then it will acquire them and return `TRUE`. Otherwise, `trywait()` will return `FALSE` without acquiring any resource. For the sake of simplicity, it is assumed that the conditions with two or more literals are evaluated atomically.

Algorithm 1 - Code for Philosopher i (where $i \in \{1, 2, 3, 4, 5\}$)

```
var
     fork: array [1..5] of semaphores; // shared resources
     bowl: semaphore;                  // shared resource
begin
   do while (1)
      THINK;
      Enter the room;
(T1)  wait(fork[i]);
      do while (not(trywait(bowl, fork[i+1 mod 5]))
                or the spaghetti is cold)
(T2)     if (trywait(bowl)
                and the spaghetti is cold) then
(T3)        signal(fork[i]);
            Go to the microwave;
            Heat up spaghetti;
            Go back to table;
(T4)        wait(fork[i]);
(T5)        signal(bowl);
         end if;
(T6̄)  loop;
      Serve spaghetti;
(T7)  signal(bowl);
      EAT;
(T8)  signal(fork[i], fork[i+1 mod 5]);
      Leave the room;
   loop;
```

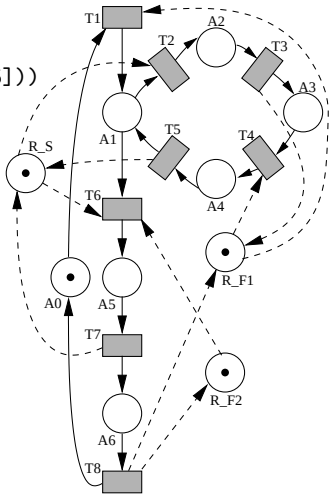


Fig. 1. Philosopher 1.

Figure 1 depicts the net for algorithm 1, with $i = 1$, after abstracting the relevant information from a RAS perspective. Figure 2 renders the composition of the five philosopher nets via fusion of the common shared resources. Note that if we remove the dashed arcs from figure 2, then we can see five disjoint strongly connected state machines plus six isolated places.

The five state machines represent the control flow for each philosopher. Every state machine is composed of seven states (each state being represented by a place). Tokens in a state machine represent concurrent processes/threads which share the same control flow. In this case, the unique token in each machine is located at the so-called *idle place*. This means that, at the initial state, every philosopher is thinking (outside the room). In general, the idle place can be seen

as a mechanism which enforces a structural bound: the number of concurrent *active threads* (i.e. non-idle) is limited. Here, at most one philosopher of type i can be inside the room, for each $i \in \{1, 2, 3, 4, 5\}$.

The six isolated places are called *resource places*. A resource place represents a certain resource type, and the number of tokens in it represents the quantity of free instances of that resource type. In this case, every resource place is monomarked. Thus, at the initial state there is one fork of type i, for every $i \in \{1, 2, 3, 4, 5\}$, plus one bowl of spaghetti (modelled by way of the resource place at the centre of the figure).

Finally, the dashed arcs represent the acquisition or release of resources by the active threads when they change their execution state. Every time a transition is fired, the total amount of resources available is altered. Please note, however, that moving one isolated token of a state machine (by firing its transitions) until the token reaches back the idle state, leaves the resource places marking unaltered. Thus, the resource usage is conservative.

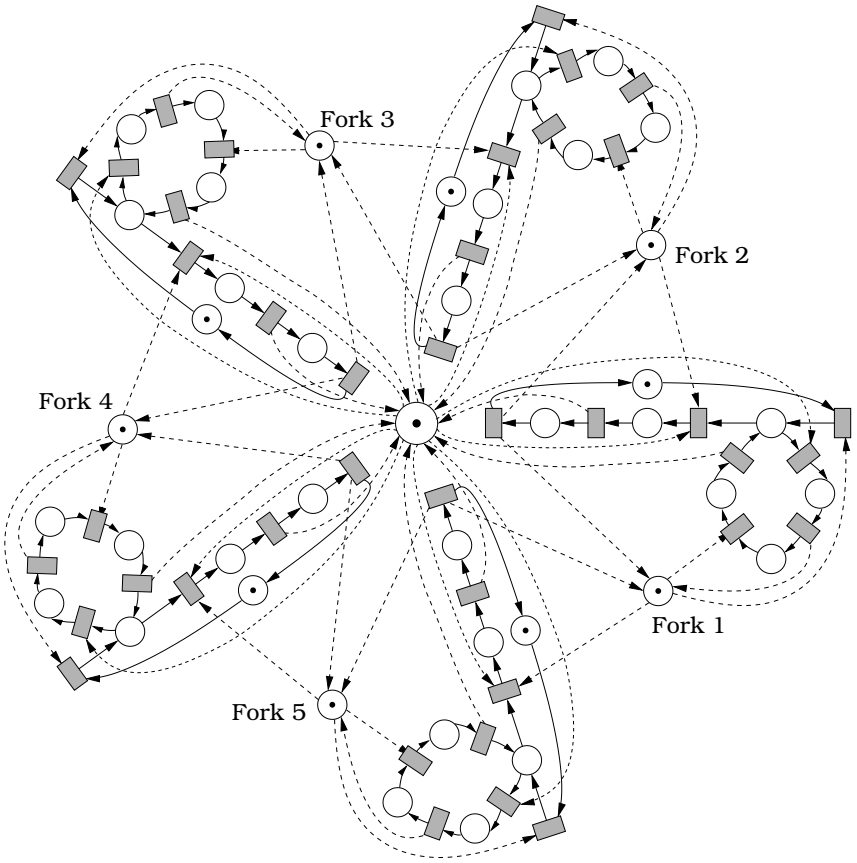


Fig. 2. The dining philosophers are thinking. Arcs from/to P_R are dashed for clarity.

At this point, we will discuss some capabilities that (in our humble opinion) a RAS model should have so as to support the modelling of concurrent programs.

Although acyclic sequential state machines are rather versatile as models for sequential processes in the context of FMS (as the success of the S^3PR and S^4PR classes prove), this is clearly too constraining even for very simple software systems. Considering Böhm and Jacopini's theorem [18], however, we can assume that every non-structured sequential program can be refactored into a structured one using `while-do` loops. Meanwhile, calls to procedures and functions can be substituted by way of inlining techniques. Let us also remind that `fork`/`join` operations can also be unfolded into isolated concurrent sequential processes, as evidenced in [9]. As a result, we can restrict process models to state machines in which decisions and iterations (in the form of `while-do` loops) are supported, but not necessarily every kind of internal cycle.

Another significant difference between FMS and software systems from a RAS perspective is that resources in the latter are not necessarily physical (e.g., a file) but can also be logical (e.g., a semaphore). This has strong implications in the degree of freedom allowed for allocating those resources: we will return to this issue a little later.

In this domain, a resource is an object that is shared among concurrent processes/threads and must be used in mutual exclusion. Since the number of resources is limited, the processes will compete for the resource and will use it in a non-preemptive way. This particular allocation scheme can be imposed by the resources' own access primitives, which may be blocking. Otherwise, the resource can be protected by a binary semaphore/mutex/lock (if there is only one instance of that resource type) or by a counting semaphore (multiple instances). Note that this kind of resources can be of assorted nature (e.g., shared memory locations, storage space, database table rows) but the required synchronization scheme is inherently similar.

On the other side, it is well-known that semaphores used in that aim can be also seen as non-preemptive resources which are used in a conservative way. For instance, a counting semaphore that limits the number of connections to a database can be interpreted in that way from a RAS point of view. Here processes will wait for the semaphore when attempting to establish a database connection, and will release it when they decide to close the aforementioned connection.

However, semaphores also perform a relevant role as an interprocess signaling facility, which can also be a source of deadlocks. In this work, our goal is the study of the resource allocation problem, so this functionality is out of scope. We propose fixing deadlock problems due to resource allocation issues firstly, and later apply other techniques for amending those due to message passing.

Due to their versatility, semaphore primitives are interesting for studying how resources can be allocated by a process/thread. For instance, XSI semaphores (also known as System V semaphores) have a multiple wait primitive (`semop` with `sem_op<0`). An example of multiple resource allocation appears in algorithm 1. Besides, an XSI semaphore can be decremented atomically in more than one. Both POSIX semaphores (through `sem_trywait`) and XSI semaphores (through

`semop` with `sem_op<0` and `sem_flag=IPC_NOWAIT`) have a non-blocking wait primitive. Again, algorithm 1 could serve as an example. Finally, XSI semaphores also feature inhibition mechanisms (through `semop` with `sem_op=0`), i.e. processes can wait for a zero value of the semaphore.

As we suggested earlier, the fact that resources in software engineering do not always have a physical counterpart is a very peculiar characteristic with consequences. In this context, processes do not only consume resources but also can *create* them. A process will destroy the newly created resources before its termination. For instance, a process can create a shared memory variable (or a service!) which can be allocated to other processes/threads. Hence the resource allocation scheme is no longer *first-acquire-later-release*, but it can be the other way round too. Nevertheless, all the resources will be used in a conservative way by the processes (either by a create-destroy sequence or by a wait-release sequence). As a side effect, and perhaps counterintuitively, there may not be free resources during the system startup (as they still must be created), yet being the system live.

Summing up, for successfully modelling RAS in the context of software engineering, a Petri net model should have at least the following abstract properties:

1. The control flow of the processes should be represented by state machines with support for decisions (`if-then-else` blocks) and nested internal cycles (`while-do` blocks).
2. There can be several resource types and multiple instances of each one.
3. State machines can have multiple tokens (representing concurrent threads).
4. Processes/threads use resources in a conservative way
5. Acquisition/release arcs can have non-ordinary weights (e.g., a semaphore value can be atomically incremented/decremented in more than one unit)
6. Atomic multiple acquisition/release operations must be allowed
7. Processes can have decisions dependent of the allocation state of resources (due to the non-blocking wait primitives, as in figure 2)
8. Processes can lend resources. As a side effect, there could exist processes that depend on resources which must be created/lent by other processes (hence they cannot finish if executed in isolation)

3 PC²R nets

In this section, we will present a new Petri net class, which fulfills the requirements advanced in section 2: the class of Processes Competing for Conservative Resources (PC^{2}R). This class generalizes other subclasses of the S^{n}PR family while respecting the design philosophy on these. Hence, previous results are still valid in the new framework. However, PC^{2}R nets can deal with more complex scenarios which were not yet addressed from the domain of S^{n}PR nets.

Definition 1 presents a subclass of state machines which is used for modelling the control flow of the processes in isolation. Iterations are allowed, as well as decisions within internal cycles, in such a way that the control flow of structured

programs can be fully supported. Non-structured processes can still be refactored into them as discussed in Section 2.

Definition 1. *An* iterative state machine $\mathcal{N} = \langle \{p_0\} \cup P, T, C \rangle$ *is a strongly connected state machine such that either every cycle contains* p_0 *or* P *can be partitioned into two subsets* P_1, P_2*, with a place* $p \in P_2$ *such that:*

1. *The subnet generated by* $\langle \{p\} \cup P_1, {}^\bullet P_1 \cup P_1{}^\bullet \rangle$ *is a strongly connected state machine in which every cycle contains* p*, and*
2. *The subnet generated by* $\langle \{p_0\} \cup P_2, {}^\bullet P_2 \cup P_2{}^\bullet \rangle$ *is an iterative state machine.*

In figure 1, if we remove the resource places R_F1, R_F2 and R_S then we obtain an iterative state machine, with $P_1 = \{A2, A3, A4\}$, $P_2 = \{A1, A5, A6\}$, $p_0 = A0$ and $p = A1$. The definition of iterative state machines is instrumental for introducing the class of PC^2R nets.

PC^2R nets are modular models. Two PC^2R nets can be composed into a new PC^2R model via fusion of the common shared resources. Please note that a PC^2R net can simply be one process modelled by an iterative state machine along with the set of resources it uses. Hence the whole net model can be seen as a composition of the modules for each process. We will formally define the class in the following:

Definition 2. *Let* $I_\mathcal{N}$ *be a finite set of indices. A* PC^2R *is a connected generalized pure P/T net* $\mathcal{N} = \langle P, T, C \rangle$ *where:*

1. $P = P_0 \cup P_S \cup P_R$ *is a partition such that: (a)* [idle places] $P_0 = \{p_{0_1}, ..., p_{0_{|I_\mathcal{N}|}}\}$*; (b)* [process places] $P_S = P_1 \cup ... \cup P_{|I_\mathcal{N}|}$*, where* $\forall i \in I_\mathcal{N}$*:* $P_i \neq \emptyset$ *and* $\forall i, j \in I_\mathcal{N}$*:* $i \neq j$*,* $P_i \cap P_j = \emptyset$*; (c)* [resource places] $P_R = \{r_1, ..., r_n\}, n > 0$.
2. $T = T_1 \cup ... \cup T_{|I_\mathcal{N}|}$*, where* $\forall i \in I_\mathcal{N}, T_i \neq \emptyset$*, and* $\forall i, j \in I_\mathcal{N}, i \neq j, T_i \cap T_j = \emptyset$.
3. *For all* $i \in I_\mathcal{N}$ *the subnet generated by restricting* $\mathcal{N}$ *to* $\langle \{p_{0_i}\} \cup P_i, T_i \rangle$ *is an iterative state machine.*
4. *For each* $r \in P_R$*, there exists a unique minimal p-semiflow associated to* r*,* $Y_r \in \mathbb{N}^{|P|}$*, fulfilling:* $\{r\} = \|Y_r\| \cap P_R, (P_0 \cup P_S) \cap \|Y_r\| \neq \emptyset$*, and* $Y_r[r] = 1$.
5. $P_S = \bigcup_{r \in P_R} (\|Y_r\| \setminus \{r\})$.

Please note that the support of the Y_r p-semiflows (point 4 of definition 2) may include P_0: this is new with respect to S^4PR nets. Such a resource place r is called a *lender* resource place. If r is a lender, then there exists a process which creates (*lends*) instances of r. In our model, processes can start their execution creating resource instances, but *before* acquiring any other resource. Otherwise, it could happen that the support of a minimal p-semiflow would contain more than one resource place (thus infriging condition 4 of definition 2).

The class supports iterative processes, multiple resource acquisitions, non-blocking wait operations and resource lending. Inhibition mechanisms are not natively supported (although some cases can still be modelled with PC^2R nets).

The next definition generalizes the notion of acceptable initial marking introduced for the S^4PR class. In software systems all processes/threads are initially

inactive and start from the same point (the `begin` operation). Hence, all of the corresponding tokens are in the idle place in the initial marking (the process places being therefore empty). Note that lender resource places may be empty for an acceptable initial marking. Figure 2 shows a P^2CR net with an acceptable initial marking which does not belong to the S^4PR class.

Definition 3. *Let $\mathcal{N} = \langle P_0 \cup P_S \cup P_R, T, C \rangle$ be a PC^2R. An initial marking m_0 is acceptable for $\mathcal{N}$ iff $||m_0|| = P_0 \cup P_R$ and $\forall p \in P_S, r \in P_R \ : \ Y_r^T \cdot m_0 \geq Y_r[p]$.*

4 Some transformations and related classes

In [19], we introduced a new class of Petri net models for RAS, called SPQR (Systems of Processes Quarreling over Resources). SPQR nets feature an appealing syntactical simplicity and expressive power though they are very challenging from an analytical point of view. They can be roughly described as RAS nets in which the process subnets are acyclic and the processes can lend resources in any possible (conservative) manner. Every PC^2R can be transformed into a Structurally Bounded SPQR net (SB SPQR net).

The transformation rule is based on the idea of converting every while-do block in an acyclic process which is activated by a lender resource place. This lender place gets marked once the thread reaches the while-do block. The token is removed at the exit of the iteration. This transformation must be applied starting by the most intern loops, proceeding in decreasing nesting order. Figure 3 depicts the transformation rule. The rule preserves the language accepted by the net (and thus liveness) since it basically consists in the addition of a implicit place (place $P1$ in the right hand net of figure 3, since R_P1 can be seen as a renaming of $P1$ in the left hand net).

Figure 4 illustrates the transformation of the net of example 1 but restricted to two philosophers into the corresponding SB SPQR.

Thanks to such transformations, the SB SPQR class can express the widest range of systems in the Sequential RAS Petri net family. Figure 5 introduces the inclusion relations between a variety of Petri net classes for Sequential RAS.

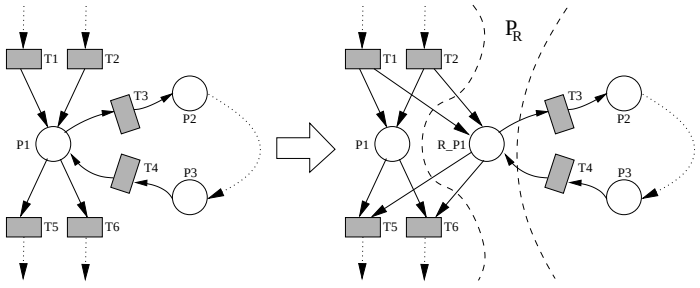


Fig. 3. Transforming PC^2Rs into SB SPQRs: From iterative to acyclic processes

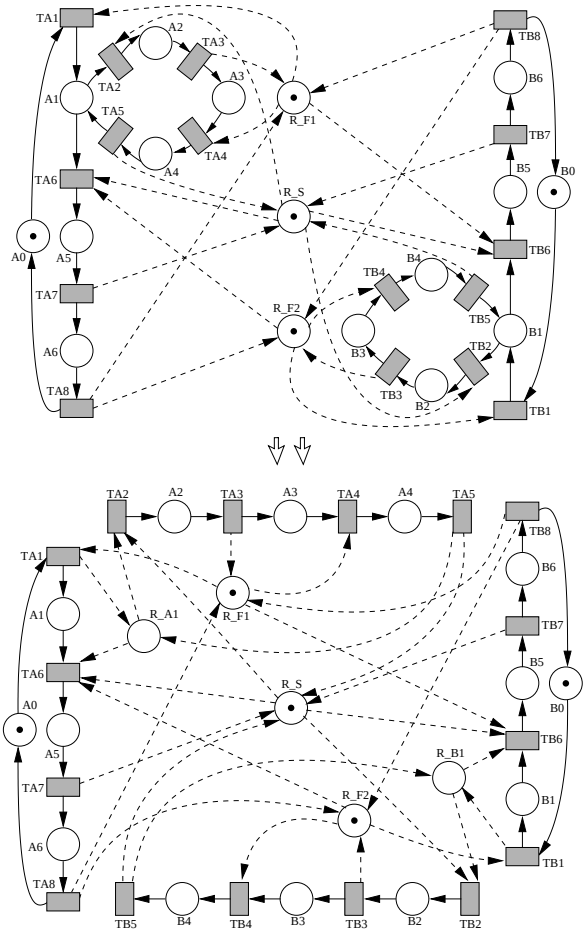


Fig. 4. From PC^{2}R to SB SPQR: Two pragmatic dining philosophers

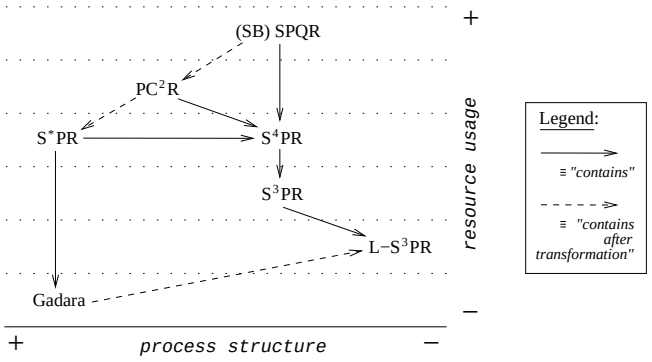


Fig. 5. Inclusion relations between Petri net classes for RAS

5 Some bad properties through examples

The bad news about the discussion in sections 2 and 3 is that siphon-based control techniques for RAS do not work in general for concurrent software, even ignoring (i.e., not using) the *resource lending* feature introduced by PC^{2}R nets.

Let us have a look back at example 1 and its related algorithm 1. It is not difficult to see that, if every philosopher enters the room, sits down and picks up the fork on the left of himself, the philosophers will be trapped in a livelock. Every philosopher can eventually take the bowl of spaghetti and heat it up in the microwave. This pattern can be repeated infinitely, but it is completely useless, since no philosopher will ever be able to have dinner.

This behaviour is obviously reflected in the corresponding net representation at figure 2. Let us construct a firing sequence σ containing only the first transition of each state machine (i.e., the output transition of its idle place). The firing order of these transitions is irrelevant. Now let us fire such a sequence, and the net falls in a livelock. The internal cycles are still firable in isolation, but no idle place can ever be marked again. Unfortunately, the net has several bad siphons, but none of them is empty or insufficiently marked in the livelock. In other words, for every reachable marking in the livelock, there exist output transitions of the siphons which are firable. As a result, the siphon-based non-liveness characterization for earlier net classes (such as S^{4}PR [10]) is not sufficient in the new framework.

A similar pattern can be observed in the upper net of figure 4. There exist three bad siphons, which are $D_1 = \{A2, A3, A4, A5, A6, B2, B4, B5, B6, R\_F2, R\_S\}$, $D_2 = \{A2, A4, A5, A6, B2, B3, B4, B5, B6, R\_F1, R\_S\}$ and $D_3 = \{A2, A4, A5, A6, B2, B4, B5, B6, R\_F1, R\_F2, R\_S\}$. Besides, every transition in the set $\Omega = \{TA2, TA3, TA4, TA5, TB2, TB3, TB4, TB5\}$ is an output transition of D_1, D_2 and D_3. After firing $TA1$ and $TB1$ from the initial marking, the state $A1 + B1 + R\_S$ is reached. This marking belongs to a livelock with other six markings. The reader can check that, unfortunately, there exists a firable transition in Ω for every marking in the livelock. A similar phenomenon can be observed for the SB SPQR net at the bottom of figure 4.

In general, livelocks are not a new phenomenon in the context of Petri net models for RAS. Even for $L - S^3PR$ nets, which are the simplest models in the family, deadlock freeness does not imply liveness [20]. However, deadlocks and livelocks always could be related to the existence of a siphon which was 'dry'. Unfortunately, this no longer holds. Another well-known result for simpler subclasses was that liveness equalled reversibility for nets with acceptable initial markings. For PC^{2}R, this is also also untrue, as figure 6 proves.

We believe that the transformation of PC^{2}R nets into SB SPQR can be useful to understand the phenomena from a structural point of view. Intuitively speaking, the concept of *lender resource* seems a simple yet powerful instrument which still remains to be fully explored. Still, SB SPQRs can present very complex behaviour. For instance, acceptably marked SB SPQR nets do not even hold the directness property [21] (which e.g. was true for S^{4}PR nets). Figure 7 shows a marked net which has no home states in spite of being live. This and

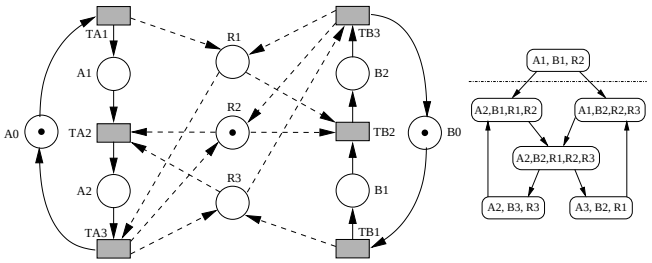


Fig. 6. An acceptably marked PC^2R which is live but not reversible

other properties are profoundly discussed (along with their implications) in a previous work [19].

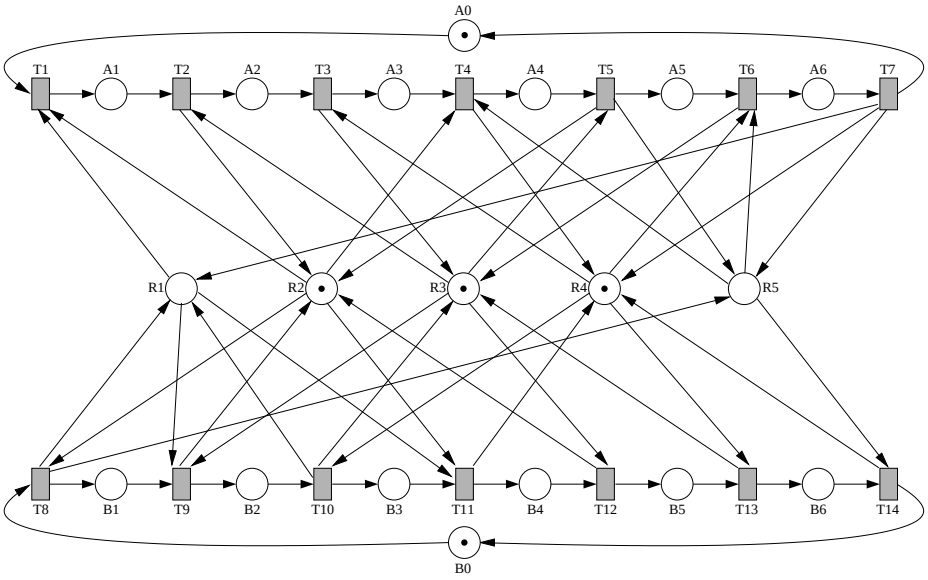


Fig. 7. A marked SB SPQR which is live but has no home states

6 Conclusion and future work

Although there exist a variety of Petri net classes for RAS, many of these definition efforts have been directed to obtain powerful theoretical results for the analysis and synthesis of this kind of systems. Nevertheless, we believe that the process of abstraction is a central issue in order to have useful models from a real-world point of view, and therefore requires careful attention. In this work,

we have followed that path and constructed a requirements list for obtaining an interesting Petri net subclass of RAS models applied to the software engineering domain. Considering that list, we defined the class of PC^2R nets, which fulfills those requirements while respecting the design philosophy on the RAS view of systems. We also introduced some useful transformation and class relations so as to locate the new class among the myriad of previous models. Finally we observed that the problem of liveness in the new context is non-trivial and presented some cases of bad behaviour which will be subject of subsequent work.

A Petri Nets: Basic definitions

A *place/transition net* (P/T net) is a 3-tuple $\mathcal{N} = \langle P, T, W \rangle$, where W is a total function $W : (P \times T) \cup (T \times P) \to \mathbb{N}$, being P, T non empty, finite and disjoint sets. Elements belonging to the sets P and T are called respectively *places* and *transitions*, or generally nodes. P/T nets can be represented as a directed bipartite graph, where places (transitions) are graphically denoted by circles (rectangles): let $p \in P$, $t \in T$, $u = W(p,t)$, $v = W(t,p)$, there is a directed arc, labelled u (v), beginning in p (t) and ending in t (p) iff $u \neq 0$ ($v \neq 0$).

The *preset* (*poset*) or set of input (output) nodes of a node $x \in P \cup T$ is denoted by ${}^\bullet x$ ($x^\bullet$), where ${}^\bullet x = \{y \in P \cup T \mid W(y,x) \neq 0\}$ ($x^\bullet = \{y \in P \cup T \mid W(x,y) \neq 0\}$). The preset (poset) of a set of nodes $X \subseteq P \cup T$ is denoted by ${}^\bullet X$ ($X^\bullet$), where ${}^\bullet X = \{y \mid y \in {}^\bullet x, x \in X\}$ ($X^\bullet = \{y \mid y \in x^\bullet, x \in X\}$

An *ordinary P/T net* is a net with unitary arc weights (i.e., W can be defined as a total function $(P \times T) \cup (T \times P) \to \{0,1\}$). If the arc weights can be non-unitary, the P/T net is also called *generalized*. A *state machine* is an ordinary net such that for every transition $t \in T$, $|{}^\bullet t| = |t^\bullet| = 1$. An *acyclic state machine* is an ordinary net such that for every transition $t \in T$, $|{}^\bullet t|, |t^\bullet| \leq 1$, and there is no circuit in it.

A self-loop place $p \in P$ is a place such that $p \in p^{\bullet\bullet}$. A *pure P/T net* (also self-loop free P/T net) is a net with no self-loop places. In pure P/T nets, the net can be also defined by the 3-tuple $\mathcal{N} = \langle P, T, C \rangle$, where C is called the *incidence matrix*, $C[p,t] = W(p,t) - W(t,p)$. Nets with self-loop places can be easily transformed into pure P/T nets without altering most significant behavioural properties, such as liveness, as shown in figure 8.

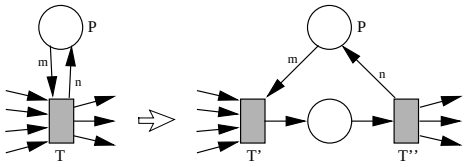


Fig. 8. Removing self-loop places

A *p-flow* is a vector $Y \in \mathbb{Z}^{|P|}$, $Y \neq \mathbf{0}$, which is a left annuler of the incidence matrix, $Y \cdot C = \mathbf{0}$. The support of a p-flow is denoted $\|Y\|$, and its places are said to be covered by Y. A *p-semiflow* is a non-negative p-flow, i.e. a p-flow such that $Y \in \mathbb{N}^{|P|}$. The P/T net $\mathcal{N}$ is *conservative* iff every place is covered by a p-semiflow. A *minimal p-semiflow* is a p-semiflow such that the g.c.d of its non-null components is one and its support $\|Y\|$ is not an strict superset of the support of another p-semiflow.

A set of places $D \subseteq P$ is a *siphon* iff every place $p \in {}^\bullet D$ holds $p \in D^\bullet$. The support of a p-semiflow is a siphon but the opposite does not hold in general.

Let $\mathcal{N} = \langle P, T, W \rangle$ be a P/T net, and let $P' \subseteq P$ and $T' \subseteq T$, where $P', T' \neq \emptyset$. The P/T net $\mathcal{N}' = \langle P', T', W' \rangle$ is the subnet generated by P', T' iff $W'(x, y) \Leftrightarrow W(x, y)$, for every pair of nodes $x, y \in P' \cup T'$.

A *marking* m of a P/T net $\mathcal{N}$ is a vector $\mathbb{N}^{|P|}$, assigning a finite number of marks $m[p]$ (called *tokens*) to every place $p \in P$. Tokens are represented by black dots within the places. The *support* of a marking, $\|m\|$, is the set of places which are marked in m, i.e. $\|m\| = \{p \in P \mid m[p] \neq 0\}$. We define a *marked P/T net* (also P/T net system) as the pair $\langle \mathcal{N}, m_0 \rangle$, where $\mathcal{N}$ is a P/T net, and m_0 is a marking for $\mathcal{N}$, also called *initial marking*. $\mathcal{N}$ is said to be the structure of the system, while m_0 represents the system state.

Let $\langle \mathcal{N}, m_0 \rangle$ be a marked P/T net. A transition $t \in T$ is *enabled* (also *firable*) iff $\forall p \in {}^\bullet t \ : \ m_0[p] \geq W(p, t)$, which is denoted by $m_0[t\rangle$. The *firing* of an enabled transition $t \in T$ changes the system state to $\langle \mathcal{N}, m_1 \rangle$, where $\forall p \in P \ : \ m_1[p] = m_0[p] + C[p, t]$, and is denoted by $m_0[t\rangle m_1$. A *firing sequence* σ from $\langle \mathcal{N}, m_0 \rangle$ is a non-empty sequence of transitions $\sigma = t_1\, t_2 \ldots t_k$ such that $m_0[t_1\rangle m_1[t_2\rangle \ldots m_{k-1}[t_k\rangle$. The firing of σ is denoted by $m_0[\sigma\rangle t_k$. A marking m is *reachable* from $\langle \mathcal{N}, m_0 \rangle$ iff there exists a firing sequence σ such that $m_0[\sigma\rangle m$. The *reachability set* $RS(\mathcal{N}, m_0)$ is the set of reachable markings, i.e. $RS(\mathcal{N}, m_0) = \{m \mid \exists\ \sigma \ : \ m_0[\sigma\rangle m\}$.

A transition $t \in T$ is *live* iff for every reachable marking $m \in RS(\mathcal{N}, m_0)$, $\exists m' \in RS(\mathcal{N}, m)$ such that $m'[t\rangle$. The system $\langle \mathcal{N}, m_0 \rangle$ is *live* iff every transition is live. Otherwise, $\langle \mathcal{N}, m_0 \rangle$ is *non-live*. A transition $t \in T$ is *dead* iff there is no reachable marking $m \in RS(\mathcal{N}, m_0)$ such that $m[t\rangle$. The system $\langle \mathcal{N}, m_0 \rangle$ is a *total deadlock* iff every transition is dead, i.e. no transition is firable. A *home state* m_k is a marking such that it is reachable from every reachable marking, i.e. $\forall m \in RS(\mathcal{N}, m_0) \ : \ m_k \in RS(\mathcal{N}, m)$. The net system $\langle \mathcal{N}, m_0 \rangle$ is *reversible* iff m_0 is a home state.

References

1. Lautenbach, K., Thiagarajan, P.S.: Analysis of a resource allocation problem using Petri nets. In Syre, J.C., ed.: Proc. of the 1st European Conf. on Parallel and Distributed Processing, Toulouse, Cepadues Editions (1979) 260–266
2. Colom, J.M.: The resource allocation problem in flexible manufacturing systems. In van der Aalst, W-M-P. and Best, E., ed.: Proc. of the 24th Int. Conf. on Applications and Theory of Petri Nets. Volume 2679 of LNCS., Eindhoven, Netherlands, Springer–Verlag (June 2003) 23–35

3. Li, Z.W., Zhou, M.C.: Deadlock Resolution in Automated Manufacturing Systems: A Novel Petri Net Approach. Springer, New York, USA (2009)
4. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Computing Surveys **3**(2) (1971) 67–78
5. Reveliotis, S.A., Lawley, M.A., Ferreira, P.M.: Polynomial complexity deadlock avoidance policies for sequential resource allocation systems. IEEE Transactions on Automatic Control **42**(10) (1997) 1344–1357
6. Fanti, M.P., Maione, B., Mascolo, S., Turchiano, B.: Event-based feedback control for deadlock avoidance in flexible production systems. IEEE Transactions on Robotics and Automation **13**(3) (1997) 347–363
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
8. Ezpeleta, J., Colom, J.M., Martínez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation **11**(2) (April 1995) 173–184
9. Ezpeleta, J., Recalde, L.: A deadlock avoidance approach for non–sequential resource allocation systems. IEEE Transactions on Systems, Man and Cybernetics. Part–A: Systems and Humans **34**(1) (January 2004)
10. Tricas, F., García-Valles, F., Colom, J.M., Ezpeleta, J.: A Petri net structure-based deadlock prevention solution for sequential resource allocation systems. In: Proc. of the 2005 Int. Conf. on Robotics and Automation (ICRA), Barcelona, Spain, IEEE (April 2005) 272–278
11. Park, J., Reveliotis, S.A.: Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings. IEEE Transactions on Automatic Control **46**(10) (2001) 1572–1583
12. Ezpeleta, J., Tricas, F., García-Vallés, F., Colom, J.M.: A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. IEEE Transactions on Robotics and Automation **18**(4) (August 2002) 621–625
13. Xie, X., Jeng, M.D.: ERCN-merged nets and their analysis using siphons. IEEE Transactions on Robotics and Automation **29**(4) (1999) 692–703
14. Jeng, M.D., Xie, X.L., Peng, M.Y.: Process nets with resources for manufacturing modeling and their analysis. IEEE Transactions on Robotics **18**(6) (2002) 875–889
15. Hu, H.S., Zhou, M.C., Li, Z.W.: Liveness enforcing supervision of video streaming systems using non-sequential Petri nets. IEEE Transactions on Multimedia **11**(8) (December 2009) 1446–1456
16. Wang, Y., Liao, H., Reveliotis, S., Kelly, T., Mahlke, S., Lafortune, S.: Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In: Proc. of the 49th IEEE Conf. on Decision and Control, Atlanta, Georgia, USA, IEEE (December 2009) 4971–4976
17. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (1978) 666–677
18. Harel, D.: On folk theorems. Communications of the ACM **23**(7) (1980) 379–389
19. López-Grao, J.P., Colom, J.M.: Lender processes competing for shared resources: Beyond the S^4PR paradigm. In: Proc. of the 2006 Int. Conf. on Systems, Man and Cybernetics, IEEE (October 2006) 3052–3059
20. García-Vallés, F.: Contributions to the structural and symbolic analysis of place/transition nets with applications to flexible manufacturing systems and asynchronous circuits. PhD thesis, University of Zaragoza, Zaragoza (April 1999)
21. Best, E., Voss, K.: Free choice systems have home states. Acta Informatica 21 (1984) 89–100

Nets-Within-Nets Paradigm and Grid Computing

Marco Mascheroni, Fabio Farina

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca, 336, I-20126 Milano (Italy)**

Abstract. Grid is one of the most effective new paradigms in large scale distributed computing. Only recently Petri nets have been adopted as a formal modeling framework for describing the specific aspects of the Grid. In this paper we describe a Grid tool for High Energy Physics data analysis, and we show how modeling its architecture with nets-within-nets has led us to identify and solve a number of defects affecting the current implementation.

1 Introduction

In the last decade the Grid computing [10, 9] approach to parallel and distributed computing has defined a new path to enable high performance and throughput applications. Grid infrastructures expose computational and storage resources provided by different computing centers as uniform families of services that can be coordinated to create large scale e-Science workflows.

Grand-challenge experiments, like those related to High Energy Physics, life-science, and environmental science adopted the Grid as the tool for implementing their software. In this paper we will consider a Grid distributed data analysis tool developed to serve the community of the Compact Muon Solenoid (CMS) [19] experiment at the CERN Large Hadron Collider (LHC) [20]. A specific software tool has been developed to analyze physics data over the Grid, so that the users are protected from the architectural complexities of the distributed infrastructure itself. This application, called CMS Remote Analysis Builder (CRAB) [7] is released as open source software and has been adopted by the physics community since 2005. Even though the code quality is being continuously improved thanks to code analyzers (e.g., lint), the overall architecture has never been validated with formal tools like Petri nets.

The aim of this work is to validate some relevant parts of the CRAB tool using nets-within-nets [23]. In this paradigm the tokens of a Petri net can be Petri nets themselves. As we will see, the hierarchical structure of the system components is particularly suited for investigation with this formal framework. The Renew tool [17] has been chosen as modeling platform, as it is the only nets-within-nets tool that is mature enough to describe a real system like CRAB.

** Partially supported by MIUR (Italian Ministry of Education, University and Scientific Research)

In particular, the features of Renew used to model the system are such that the obtained model is very similar to a hypernet [2].This is a class of high level Petri nets which implements the nets-within-nets paradigm using a dynamic hierarchy, and a bounded state space [3]. As detailed in Section 4, this approach allowed us to isolate some problems in the CRAB implementation. Our approach do not cover analysis yet: modeling and simulation are the two means used to unveil these problems.

In the literature high level Petri nets have been applied to different contexts related to Grid computing technologies. Most of the works in this field focus on the usage of Petri nets as a tool for workflows specification and execution [1, 13, 11]. A different application of Petri nets to Grid is reported in [5]. Here the resources exposed by the distributed computing infrastructure are modeled directly with the aim of validating both properties like the soundness and the fairness of their sharing for a process mining workflow. As far as we know, high level Petri nets, and in particular hierarchical nets, have been applied neither to the Grid infrastructure, nor to the study of a classical Grid application pattern like the distributed data analysis.

The remainder of the paper is organized as follows: Section 2 introduces the basic notion of nets-within-nets we refer to, and the Renew tool. Section 3 describes the Grid architecture we are considering, while in Section 4 the modeling of the system and the bugs found thanks to the formal approach are presented. A discussion about the modeling choices used in our approach is made in Section 5. Finally, some conclusions are reported in Section 6.

2 The Nets-Within-Nets Paradigm and Renew

According to the nets-within-nets paradigm, the tokens of a Petri net can be structured as Petri nets themselves. This idea is due to Valk (see [21]), who defined and studied the class of *Elementary Object Nets (EOS)* in [22]. Later on, properties of EOS were studied in [15], and other classes of high level Petri nets which uses the nets-within-nets paradigm were defined, like for example [12, 2, 14, 24, 18].

In all these models a system is usually modeled as a collection of nets. One net is designated as the *system net*, the top level of the net hierarchy. All other nets are assigned to an *initial place*, a place in which they reside initially. This distribution of nets induces a hierarchy. The system evolves by moving tokens from place to place through the firing of autonomous transitions, or by synchronizing transitions between nets at different levels. The hierarchical structure of the model is usually static, but in some models there can be interactions between nets at different levels in the hierarchy which can dynamically change the hierarchy itself. For example, in hypernets a net N can be moved from a place belonging to a net A, to a place belonging to a distinct net B. The interaction between nets A and B is only possible if they are close in the hierarchy.

The development of the Renew software tool [17], a Java-based high-level Petri net simulator that provides a flexible modelling approach based on Refer-

ence nets [16], allows the use of this paradigm to model real systems. RENEW is not only a nets-within-nets editor and simulator: it allows the use high level net concepts like arc inscriptions, transition guards, and coloured tokens. However, we only use a subset of the features of RENEW. In particular, we choose to model the system with a hypernet-like model [2] (we will discuss in section 5 why the system is not a proper hypernet). The system is modeled as a collection of *net instances*. Tokens are *references* to net instances. Therefore it is possible that a net has more than one reference (token) in the system which refer to it. Arc inscriptions contain single variables. When a transition is fired tokens are bound to these variablesṪransition inscriptions may contain channel names, used by two or more nets when they need to synchronize. An *uplink* is used when a net wants to synchronize with the net above it in the hierarchy, a *downlink* is used when a net wants to synchronize with one of the reference tokens it contains.

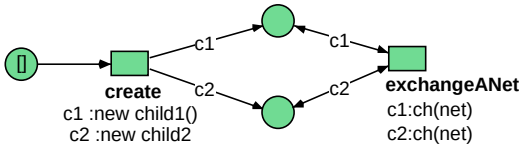
From a syntactical point of view the RENEW constructs we used in our model are the following:

- A net instance is created by a transition inscription of the form *var : new netname*, which means that the variable *var* will be assigned a new net instance of type *netname*.
- An *uplink* is specified as a transition inscription *:channelname (expr,...)*. It provides a name for the channel and an arbitrary number of parameter expressions
- A *downlink* has the form *netexpr :channelname (expr,...)* where *netexpr* is an expression that must evaluate to a net reference.

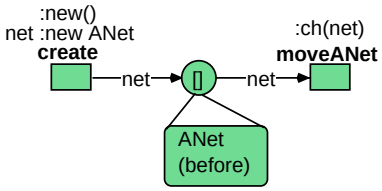
To fire a transition that has a downlink, there must be an input arc labelled with a proper variable name (*netexpr* for the previous downlink example), and this variable must evaluate to a net instance. The referenced net instance must provide an uplink with the same name and parameter count and it must be possible to bind the variables suitably so that the channel expressions evaluate to the same values on both sides. Parameters are bound to variables present in the input arcs, and then bound to the parameter in the corresponding down(up)-link. Then the transitions can fire simultaneously.

The exchange of (structured) tokens between nets, typical of hypernets, is possible by means of parameters. Figure 1 shows an example. The only transition enabled at the beginning is *create* (Figure 1(a)), which creates an empty *child1* net, and a *child2* net (Figure 1(b), and Figure 1(c) respectively). The difference between using the parenthesis or not using the parenthesis in creating a new net is that, if you use them, then the transition that is being fired must synchronize on the channel *new()* in the child net. Therefore, transition *create* in the system net synchronizes with transition *create* in the child1 net, which creates the *ANet* net. Afterwards, transitions *exchangeNet, moveANet, receiveANet* can fire, moving *ANet* to *child2*.

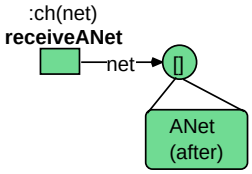
Let us notice that in our model the exchange of tokens between the two children nets, *child1* and *child2*, is made under the supervision of the system net. This means that the *system net* in some way observes the token exchange between its children.



(a) The system net



(b) The child1 net which creates a net of type *ANet* and sends it upward



(c) The child2 net which receives a net from above

Fig. 1. A simple example

3 The Application Context: Grid distributed analysis

The CMS experiment at CERN produces about 2 Petabytes of data to be stored every year, and a comparable amount of simulated data is generated. Data needs to be accessed for the whole lifetime of the experiment, for reprocessing and analysis, from a worldwide community: about 3000 collaborators from 183 institutes spread over 38 countries all around the world.

The CMS computing model uses the infrastructure provided by the Worldwide LHC Computing Grid (WLCG) Project [6] through the supporting projects EGEE, OSG and Nordugrid. Grid analysis in CMS is data driven. A prerequisite is that data is already distributed to some remote computing centers, and correspondingly published in the CMS data catalogue, so that users can discover available datasets. Parallelization is provided by splitting the analysis of large data samples into several jobs. The output data produced by the analyses are typically copied to the storage of a site and registered in the experiment specific catalogue. Small output data files are returned to the user. In the CMS experiment the CRAB tool set has been developed in order to enable physicists to perform distributed analysis over the Grid. The role of CRAB is to allow the user to run over distributed datasets the very same analysis she/he ran locally, and collect the results at the end. CRAB interacts with the distributed environment and the CMS services, hiding as much of the complexity of the system as possible. CMS community members use CRAB as a front-end which provides a thin client, and an Analysis Server which does most of the work in terms of automation, recovery, etc. with respect to the direct interactions with the Grid. The Analysis Server enables full workflow automation among differ-

ent Grid middlewares and the CMS data and workload management systems. Indeed, the main reasons behind the development for the Analysis Server are:

- automating as much as possible the whole analysis workflow;
- reducing the unnecessary human load, moving all possible actions to server side, keeping a thin and light client as the user interface;
- automating as much as possible the interactions with the Grid, performing submission, resubmission, error handling, output retrieval, post-mortem operations;
- allowing better job distribution and management;
- implementing advanced use cases for important analysis workflows

The server architecture adopts a completely modular software approach. In particular, the Analysis Server is comprised of a set of independent components (purely reactive agents) implemented as daemons and communicating asynchronously through a shared messaging service supporting the "publish & subscribe" paradigm. Most of the components are themselves implemented as multi-threaded systems, to allow a multi-user scalable system, and to avoid bottlenecks. The task analyses are completely handled during their lifetime by the server through different families of components: there are components devoted to monitoring the Grid status of the single jobs in a task, other groups of agents coordinate to manage the output retrieval and the recovery of the failed jobs by scheduling their resubmission automatically. A relevant part of the agents is designed in order to handle the submission chain of user tasks to the Grid. As the Analysis Server internal architecture is a natural candidate for being analyzed with the nets-within-nets paradigm, as aforementioned, we decided to model and study the Grid submission chain. The aim of this study is to check that the involved agents behave correctly and efficiently with respect to the foreseen submission workflow. We decided to consider the system at the component-task-job level, as it represents a good compromise between the effects perceived by the tool final users and the large number of technical details that a complete representation of the Grid would require.

4 Modeling the submission use-case

In this Section we describe in detail the process of submitting jobs to the Grid through the CRAB Analysis Server. For each relevant component of the system its net representation is discussed. In addition, the bugs that have been discovered thanks to the net models are presented with the solutions that the actual code has adopted in order to solve the issues. The CRAB analysis suite was modeled using nets in a hierarchical fashion, as shown in Figure 2. A vertical line with multiplicity n, indicates presence of a n nets in the higher one (e.g.: the CRABClient net contains from 1 to N Task nets); a horizontal dashed line indicates that the linked nets are references to the same net. In our modeling we consider one client just for the purpose of simplicity. Of course, the discussed functionalities and use cases still hold when a larger number of clients

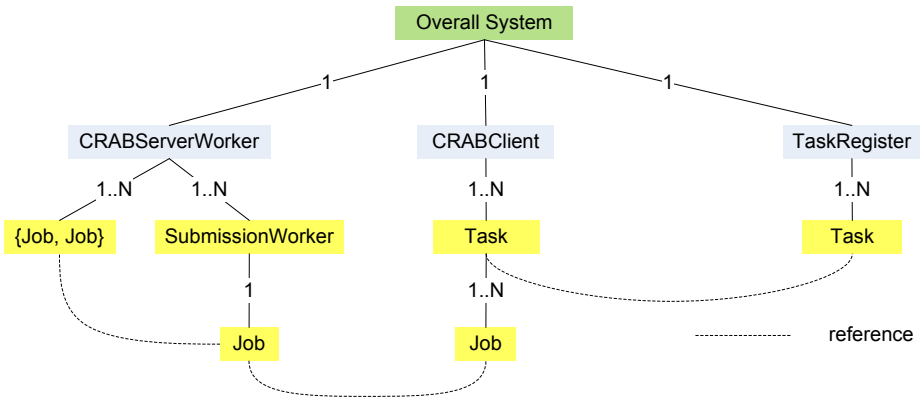


Fig. 2. The Nets hierarchy for the CRAB suite.

is considered, as the client server model assumes no direct interactions among the clients. In addition, for the use case that will be discussed, the server code separates properly the session of work for every task.

The OverallSystem net, which is the system net, contains three nets which respectively model the behavior of the client who is using the CRAB server (*CRABClient net*), the TaskRegister component which is a thread running on the CRAB server (*TaskRegister net*), and the CRABServerWorker which is also a thread running on the server (*CRABServerWorker net*). *Tasks* are the objects a client creates, and deals with. They are composed of *jobs*, the single units of work that need to be performed. The TaskRegister component is responsible for registering tasks, i.e. creating some data structures on server disks, checking if each task has all the inputs it needs to be executed, and checking if the Grid can access the proper security credentials to execute it. The CRABServerWorker component continuously receives jobs, schedules them for execution on the Grid infrastructure, and creates a SubmissionWorker thread which monitors the lifecycle of each job on the Grid. The clients interact with the server, and can initiate some operations like: submitting jobs, killing them if needed, and asking for the results.

4.1 CRABClient, Tasks, and Jobs

The first component we are going to discuss is the CRAB client, which is modeled with the net in Figure 3. This component is what enables all the action sequences that the users can do on their Grid analyses.

The first thing a client does is to create a new task on the client machine. The typical usage pairs a unique task with a CRAB analysis session. For this reason we assume that the *tasksPool* can contain a finite number of tokens. After the task has been locally created on the client machine, the client can perform a submit operation, which is of course the most important one as it starts the

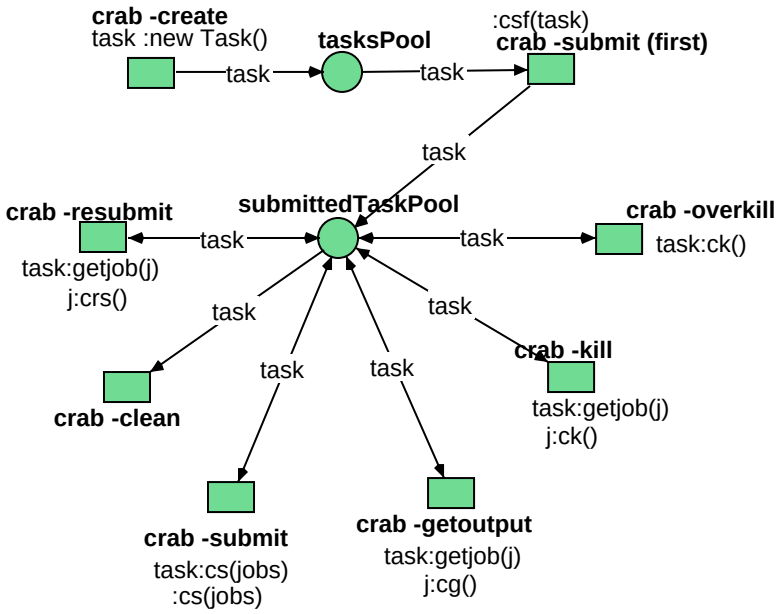


Fig. 3. The CRABClient net.

submission chain. The first time a task is submitted to the server, it is also registered by the TaskRegister component. Subsequent submits are handled directly by the CRABServerWorker component. In our model the difference between the two types of submits is modeled as two different transitions. In particular *crab -submit(first)* transition has an uplink (*:csf(task)*), which means that it must be synchronized with the upper level. As a result the task reference is copied to the TaskRegister component by the Overall System net. After creation, the main operations a user can do are submit, resubmit, kill, getoutput, and clean. All these operations require an interaction with the server, but since we have focused on the submission use case, these interactions have not been explicitly modeled. For example the *getOutput* command is modeled as an interaction between the client and the job by means of two inscriptions. Handling all the possible interactions between the actors involved in the system would have resulted in a very big model, making it impossible to describe in this paper.

A task, see Figure 4, is a bag of jobs (the system allows to collect up to 4000 jobs into a singe task) and it is a representation that CRAB uses to perform collective actions on the Grid processes. Places *notRegistered, registering, registered* of the Task net contain information about the state of a task itself. These places control the enabledness of transitions *crab -submitFirst*, and *taskRegistered*, which are respectively called by the CRABClient when a job is submitted, and by the TaskRegister component when the task has been successfully reg-

istered after a *submit first* operation. The submit transition is called when a CRABClient performs a *submit subsequent* action. In our model both *taskRegistered*, and *submit* transitions send upward two jobs through a synchronous channel, and make the job move to the submission request state.

The net representing the state of Grid jobs and their allowed actions is reported in Figure 5. This net has been modeled combining the finite state machine reported in the CRAB official documentation with the information extracted directly from the portion of code devoted to the Grid job state handling. Several transitions of this net contain uplinks, and therefore have to be synchronized with some other net. Transitions with a *:crs()* uplink (CRAB Resubmit) are transition enabled only if the job is in a state where a resubmit is possible, and are synchronized with the *crab -resubmit* transition of the CRABClient net, or the *resubmit* transition of the SubmissionWorker net. In the same way killings (channel *:ck()*), failures (channel *:f()*), submission (channel *:s()*), and output retrieving (channel *:cg()*), have to be synchronized with a correspondent transition in another net.

The integration of the documentation and the code with the formalism of the nets has allowed us to identify a bug in the way job states are modified. In particular, the net allows some transitions that are not actually activated by any event observed by the system (bug 1, b1). For example let us consider the unlabeled transition between the *sub.success* and the *cleaned* places in Figure 5: the latter denotes that a job has been abandoned because the user security credentials are expired and the Grid will not manage processes whose owner cannot be recognized. A malicious code interacting with the clients in place of the proper server could move jobs arbitrarily to this terminal state. The fix for this bug consisted in a review of the code managing the job state automata in accordance with what is stated by the presented Job net. Also the pre-conditions that allow a client to perform a *kill* request over the jobs are not granted properly (b2). This means, for example that a user could run into a condition where a failed job cannot be resubmitted as the system requires to kill it. That means the job is in a deadlock, as a failed job cannot be killed on the Grid.

4.2 TaskRegister

The TaskRegister component, shown on the left of Figure 7, duplicates the task and jobs structures that have been created at the client side and alters all the object attributes in order to localize them with respect to the running environment of the server, taking care also of security issues (like user credentials delegation) and files movement (check the existence of input). We modeled this cloning by means of the *reference semantics*: the TaskRegister component receives from the client a copy of the reference which points to the Task. The component is able to handle more tasks simultaneously thanks to a pool of threads implementing the net of Figure 7. The first transition that is fired is *submission*, which is synchronized with the transition in the system net that receives the task reference from the CRABClient. Then four operations which can fail are executed on the task. These include local modification of the task with respect to the

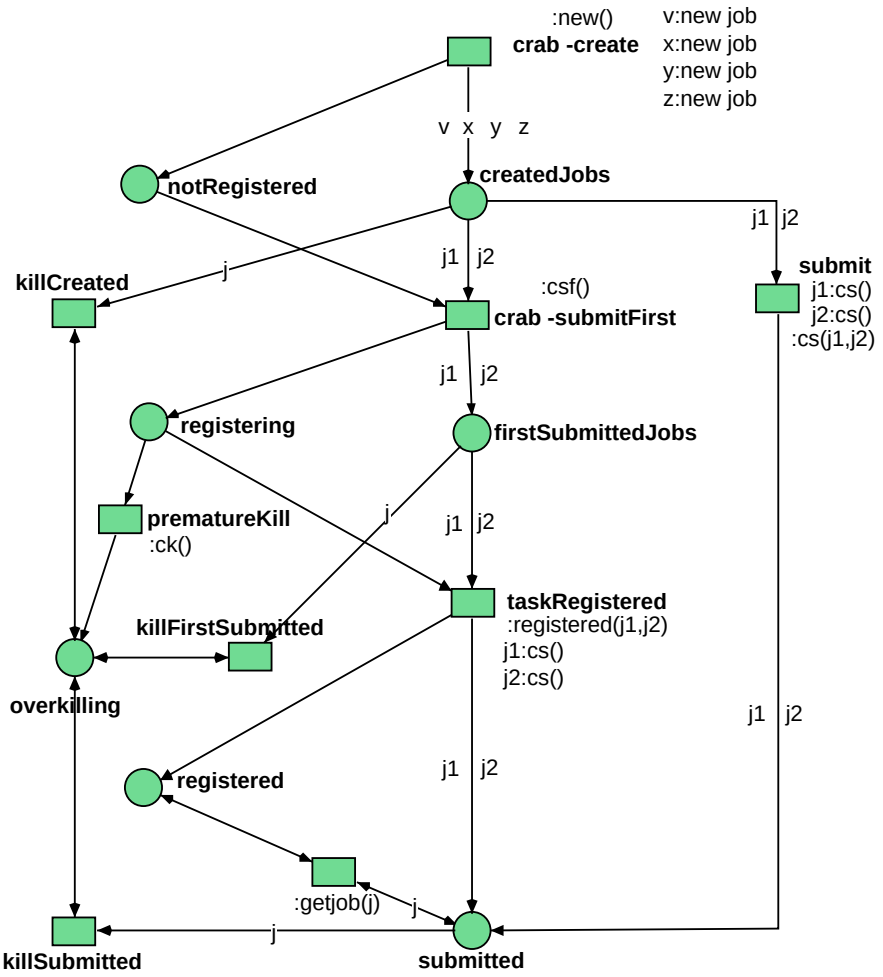


Fig. 4. The Task net. Only four jobs are considered in order to exemplify the relation with the job net.

server environment, the user's credential retrieval (also known as delegation), the setting of the server behavior according to what the credentials allow to do and, finally, the checking that the needed input files are accessible from the Grid. If the registration fails the only possible operation available is *archiveTask* which deletes the reference to the task from the task register component. If the user has the privileges to execute the jobs in the task, and if the inputs needed by the task are available, then a range of jobs is selected from the task and passed to the CRABServerWorker by firing the *toCSW* transition (again under the supervision of the system net). The modeling and the simulation of

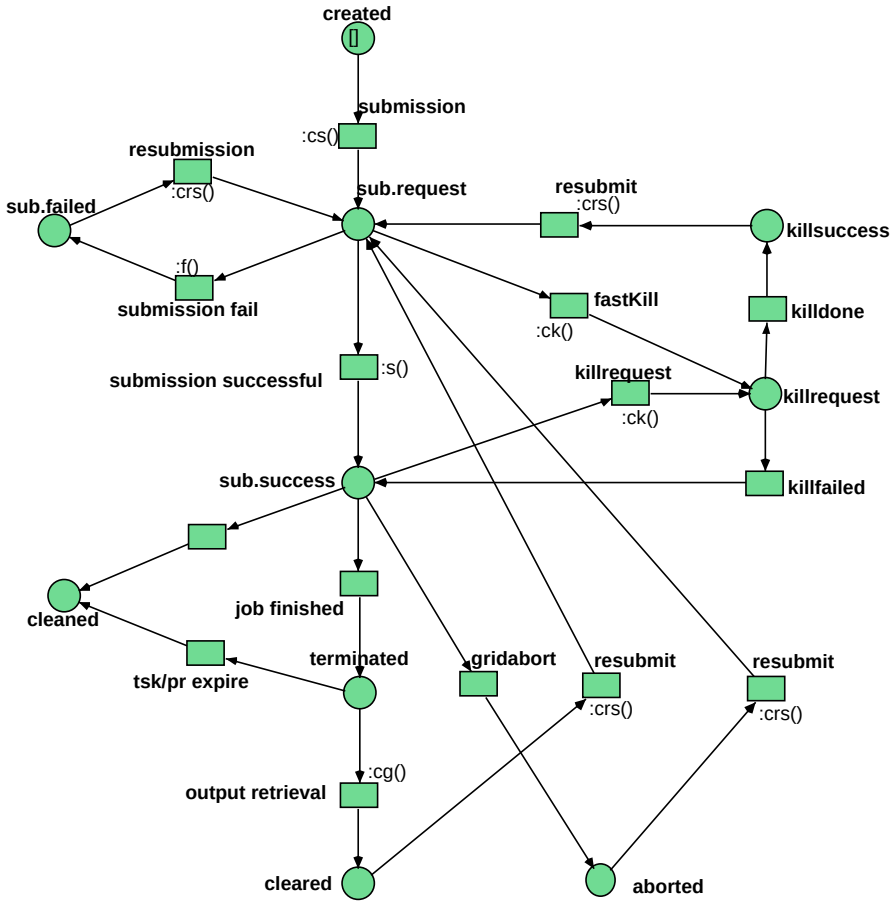


Fig. 5. The Job net.

the TaskRegister net has highlighted some relevant defects and bugs. In case of failure the TaskRegister component was not able to set properly the status of the jobs in a task to fail. This macroscopic lack in the system design implied different side effects. The server was not able to discriminate whether to retry automatically the registration process or to give up and notify the user about the impossibility to proceed (b3). In addition, the system could not tell if the registration has been attempted previously. This implies that the client transfers the input data every time a registration failure appears, with a waste of network resources (b4). This defect has been solved by introducing the proper synchronization between the *fail* transition in the component with *submission failed* in the job net. A second bug has been identified thanks to the study of the synchronization among the transitions for the client, the jobs and the TaskReg-

ister nets. In detail, the handling of the *kill* commands presents some issues. If a user requires to kill some jobs while the task is being registered, the system cannot distinguish properly which jobs have to be killed and therefore it applies an over-killing strategy by halting the whole task (b5). This happens because the code performs some sort of synchronization with the Task net instead of having rendezvous with the related transitions into the lists of killing jobs.

4.3 CRABServerWorker, and SubmissionWorkers

In our model the result of a submit operation is that the CRABServerWorker component, showed in Figure 6, receives a structured token in the place *accepted*. If the submit was the first, transition *newTaskRegistered* is fired after the task has been registered by the TaskRegister component by means of transition *toCSW*, which is synchronized with transition *newTaskRegistered* through the overall system. If the submit is not the first, the task has been already registered, therefore transition *subsequentSubmission* is fired. After receiving the range of jobs, the CRABServerWorker component schedules these jobs for the execution on the Grid infrastructure. The practical effect of this component is to break the task into lists of jobs in order to improve the performance thanks to bulk interactions with the Grid middleware. The Submission Worker thread spawned by the component monitors the actual submission process of the jobs. We have modeled this fact by creating a Submission Worker net for each one of the jobs in the list. Indeed, transition *triggerSubmissionWorker* creates a new Submission Worker assigned to the variable *sw* and synchronizes it with a transition labeled *init*. The thread is responsible both for tracking the submission to the Grid in-

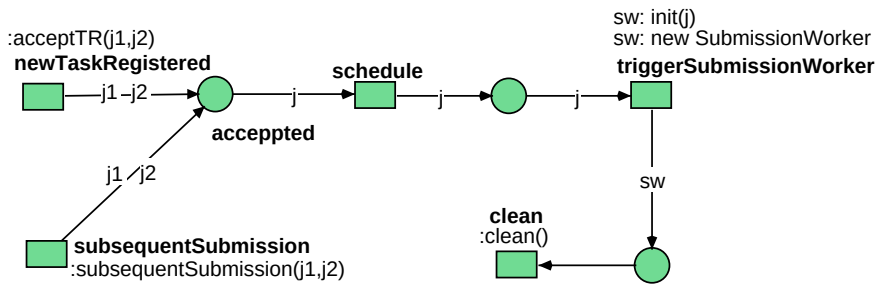


Fig. 6. The CRABServerWorker Net

frastructure, and for resubmitting jobs when a failure occurs. Failures can occur for different reasons: network communication glitches, unavailable compatible resources, etc. Some types of failures are recoverable and in those cases the Submission Worker automatically tries to resubmit the job a three times. This value can be configured but in the model we report the default case set in the code.

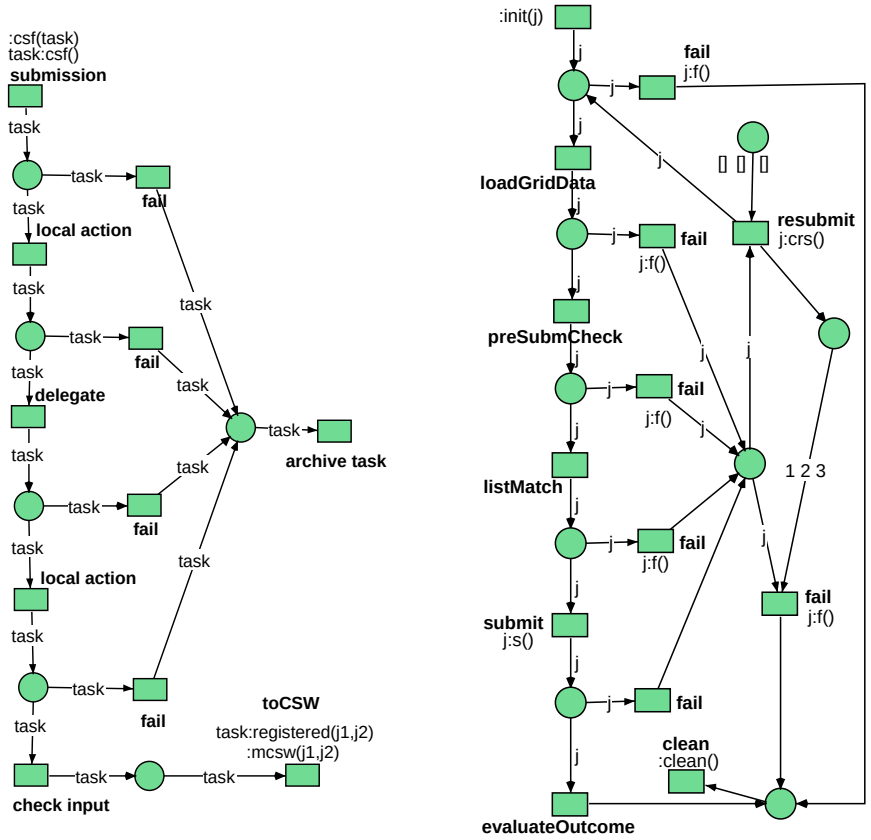


Fig. 7. TaskRegister and SubmissionWorker nets respectively

If the failure persists the job is permanently marked as failed. The net shown on the right in Figure 7 is our model of the submission worker component. The study of the synchronization between the job and the Submission Worker nets allowed us to identify another bug in the code. The *submission success* transition in the job net synchronizes with the *submit* Submission Worker's transition. This means that the CRAB Server marks the submission as successful just after the interaction with the Grid. Actually the network latencies could delay the propagation of the job failure message (b6) and, therefore, the correct rendezvous should be enacted between *submission success* and *evaluateOutcome*.

It is relevant to observe that the approach followed for the modeling of the CRAB Server submission chain is a particular case for a quite general class of Grid systems. All the Grid middlewares rely on jobs that are represented by finite state automata and that are concurrently managed by the different services involved in the Grid. In addition, the intermediate action of a broker like the CRAB Server is becoming a common pattern with the diffusion of scientific

gateways: programmatic portals that abstract the user applications from the complexities of the distributed infrastructures acting as back end.

The adoption of the nets-within-nets paradigm has provided a natural and effective way to model subtle interactions among the different net levels. It would have required a significantly greater effort to discover the same problems with a flat net approach. In the following subsection details about the process of deriving the models from the documentation and the code are given.

4.4 Details on the model derivation process

The model was derived from the code by analyzing both the official documentation and the source code of the system. The Job net is directly built from the documentation. A finite state automata which describes the Job is reported explicitly. After that, simply by using pattern matching we analyzed the source code relevant for the submission use case by searching for interaction with jobs. Each source module is modeled as a net (e.g.: CRABClient, TaskRegister, CRABServerWorker etc), and the interactions with the Job nets are modeled using the Renew uplink/downlink mechanism. A modification of the status of a job in the code is modeled as a pair of synchronized transitions in the model itself: one in the job net and one in the net that models the component changing the job status.

To ensure that the model is an accurate representation of the software, we made several task submissions with the CRAB tool and monitored the status of the jobs during the evolution. The request parameters were set up so that different behaviours of the system are tested. For example, jobs lacking of input files, job submitted by users with expired credentials, and jobs killed before the completion of task registration process are test cases that have been considered. After that, we simulated each submission on the model, taking care that the simulation of the status of the job net was consistent with the actual job status in the system. These simulations were also useful for highlighting some bugs in the code, as discussed in the previous Section.

5 Discussion

In the study we have just presented, a formal approach was used to validate a system that has already been implemented. Simulating the behavior of the system by means of a computer aided tool was what allowed us to find problems in the implementation of the CRAB server. However, another great advantage of modeling a system with formal methods is the possibility to apply *automatic* analysis techniques to extract information about the system, like invariant analysis, and model checking.

In order to apply some of these techniques, the formal model must respect specific prerequisites. For example, most algorithms for model checking a concurrent system require a bounded state space. Nets-within-nets models which satisfy this last requirement are hypernets [2] and their generalization [4], which

can both be expanded to 1-safe Petri nets [3, 18]. This expansion guarantees the possibility of applying all the analysis techniques of this well known class of Petri nets to hypernets.

The first idea was to use such a class of nets to model the CRAB server, but because of the absence of modeling limitations and verification features in Renew, and because of the high complexity of the system, we preferred to use a slighty more powerful version of hypernets. To come back to the class of hypernets, having therefore the certainty that the state space is limited, the following fixes are necessary:

- Transitions which create or delete tokens must be deleted in some way. For example, transition *crab -create* of the CrabClient net cannot create an unbounded number of tasks anymore, but an input place which contains as many tokens as the maximum number of allowed tasks must be added.
  This is not a big problem. As a matter of fact the computers disks space is limited, and consequently so are the number of tasks which can be created by a user.
- Hypernets use a *value semantics*, which means that a net cannot have two references to it. Nevertheless, in our model some transitions duplicate the references to a net. Duplication of references is somehow dangerous if the intention is to keep the state space bounded. Loosely speaking, the risk is of an uncontrolled grow of the references of a net without a corresponding deletion of these references. In our model the use of the value semantics can be achieved by deleting these duplications of references, and using simple tokens to communicate the intention to modify the referenced net.

Even though analysis of properties is not available with the current version of the model because of the issues just discussed, the more practicality of the reference semantics from a modeling point of view helped us finding several design defects in the implementation of the CRAB server. In the future we plan to restrict the model to a hypernet in order to be able to verify properties like invariants, or to do model checking [1]. In our opinion, as a first step it was important to use a powerful formalism to avoid getting lost in the details of the model, even though that meant sacrificing the analysis capabilities.

6 Conclusions

In this paper, we discuss a large scale Grid application used to perform distributed data analysis in High Energy Physics experiments. Because of the complexity of the architecture, the software tool has been modeled using the nets-within-nets paradigm in order to validate the correctness of its behavior. In particular we considered the fundamental use case of the submission of user data analysis to the Grid. Every component of the CRABServer involved in this

[1] Restricting the model to hypernet is not the only way to have a limited state space, but using hypernet you have a formal proof thank's to the 1-safe expansion

use case has been modeled in the hierarchy of the nets and compared to the behavior expected by its users.

From the analysis and simulation of the model a number of bugs and design defects emerged. This has led the developers to improve the overall quality of system implementation in the subsequent releases that the users now adopt. Two groups of bugs have been identified: bugs related to wrong coding of the expected behaviors and bugs where the specific adoption of nets-within-nets formalism has highlighted synchronization problems among the entities .

In addition, the approach followed to model the CRAB tool set has shown its generality in order to model most of the Grid applications in which an orchestration entity drives the nets representing both the finite state machines of the jobs running on the distributed infrastructure and the services exposing the resources themselves.

The class of nets used to model this system is a more powerful version of hypernets, using the reference semantics instead of the value semantics, and allowing creation/deletion of tokens. As discussed in Section 5, it is possible to restrict the model to a proper hypernet by sacrificing its readability (some places and transitions must be added). Then, by means of hypernets and their expansion to 1-safe nets, it will be possible to use all the techniques defined for the class of 1-safe nets for analyzing the system.

A plugin of Renew that allows to draw and to analyze a hypernet is being developed. We plan to use this plugin to make automatic verification of properties of the system.

References

1. Martin Alt, Andreas Hoheisel, Hans Werner Pohl, and Sergei Gorlatch. A Grid Workflow Language Using High-Level Petri Nets. In *Procs of the 6th Int. Conf. on Parallel Processing and Applied Mathematics: PPAM05*, pages 715–722, 2005.
2. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. Modelling mobility with Petri Hypernets. In *Recent Trends in Algebraic Development Techniques*, volume 3423/2005 of *Lecture Notes in Computer Science*, pages 28–44. Springer Berlin / Heidelberg, 2005.
3. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. From Petri hypernets to 1-safe nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06, Bericht 272, FBI-HH-B-272/06, 2006*, pages 23–43, June 2006.
4. Luca Bernardinello, Nicola Bonzanni, Marco Mascheroni, and Lucia Pomello. Modeling symport/antiport p systems with a class of hierarchical Petri nets. In *Membrane Computing*, volume Volume 4860/2007 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin / Heidelberg, 2007.
5. Carmen Bratosin, Wil van der Aalst, and Natalia Sidorova. Modeling Grid workflows with Coloured Petri nets. In *Procs. of the 8th Workshop on Practical Use of Coloured Petri Nets and CPN Tools: CPN 2007*, pages 67–86, 2007.
6. CERN. Worldwide LHC Computing Grid. http://lcg.web.cern.ch/lcg/public/. Accessed May, 2010.

7. Giuseppe Codispoti, Mattia Cinquilli, Alessandra Fanfani, Federica Fanzago, Fabio Farina, Carlos Kavka, Stefano Lacaprara, Vincenzo Miccio, Daniele Spiga, and Eric Vaandering. CRAB: a CMS Application for Distributed Analysis. *IEEE Transactions on Nuclear Science*, 56(5):2850–2858, 2009.
8. Jordi Cortadella and Wolfgang Reisig, editors. *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, volume 3099 of *Lecture Notes in Computer Science*. Springer, 2004.
9. Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
10. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35:37–46, 2002.
11. Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18:1115–1140.
12. Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-level nets with nets and rules as tokens. In Gianfranco Ciardo and Philippe Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2005.
13. Andreas Hoheisel and Uwe Der. Dynamic Workflows for Grid Applications. In *Procs. of the Cracow Grid Workshop 03*, page 8, 2003.
14. Michael Köhler and Berndt Farwer. Object nets for mobility. In Jetty Kleijn and Alexandre Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2007.
15. Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Cortadella and Reisig [8], pages 278–297.
16. Olaf Kummer. *Referenznetze*. Logos-Verlag, 2002.
17. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Cortadella and Reisig [8], pages 484–493.
18. Marco Mascheroni. Generalized hypernets and their semantics. In *Proceedings of the Fith International Workshop on Modelling of Objects, Components and Agents, MOCA'09, Bericht 290, 2009*, pages 87–106, September 2009.
19. The CMS Collaboration. The CMS Experiment at CERN LHC. *J. Inst.*, 3:S08004, 2008.
20. The TLS Group. The Large Hadron Collider Conceptual Design. Technical report, CERN, 1995. Preprint hep-ph/0601012.
21. Rüdiger Valk. Nets in computer organization. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume Volume 255/1987 of *Lecture Notes in Computer Science*, pages 218–233. Springer Berlin / Heidelberg, 1987.
22. Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *ICATPN*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.
23. Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Lectures on Concurrency and Petri Nets*, volume 3098/2004 of *Lecture Notes in Computer Science*, pages 819–848. Springer Berlin / Heidelberg, 2004.
24. Kees M. van Hee, Irina A. Lomazova, Olivia Oanea, Alexander Serebrenik, Natalia Sidorova, and Marc Voorhoeve. Nested nets for adaptive systems. In *ICATPN*, pages 241–260, 2006.

Verifying Reference Nets By Means of Hypernets: a Plugin for Renew

Marco Mascheroni[1], Thomas Wagner[2], and Lars Wüstenberg[2]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca, 336, I-20126 Milano (Italy)**
mascheroni@disco.unimib.it
[2] University of Hamburg,
Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. In this paper we examine ways to verify reference nets, a class of high level Petri nets supported by the Renew tool. We choose to restrict reference nets to hypernets, another nets-within-nets model more suitable for verification purposes thanks to an expansion toward 1-safe Petri nets. The contribution of the paper is the implementation of such analysis techniques by means of a Renew plugin. With this plugin it is now possible to draw, and to analyze a hypernet. The work is demonstrated by means of a simple example.

Keywords: Verification, High-level Petri nets, Reference nets, Hypernets

1 Introduction

The verification of properties of a software system has become an important part of software engineering. Especially specifications critical to the correct execution of a software system need to be verified in order to guarantee them after deployment. The problem of verification is its complexity and the effort required for it. Without proper methods the verification itself is difficult, costly and time-consuming.

In this paper we approach the general problem of verification with the help of Petri nets. The formalism is deeply rooted within established theoretical and formal methodologies, as well as being supported by a multitude of tools and analysers. Petri nets have been studied in detail and contain properties, for which established verification techniques exist. Using Petri nets the general approach is to map and translate specific software issues and properties to these Petri

** Partially supported by MIUR, and DAAD

net properties. These properties can then be verified using the known Petri net techniques. Assertions made for these can then be translated back for the software behind it.

High level nets, Petri net models enriched with additional abstraction constructs, are well suited to represent complex systems due to their high abstraction constructs. One of their problems is that verification of their properties is difficult. Properties which are computable with low-level formalisms become undecidable, and thus cannot be verified anymore in some high-level models. However, high-level formalisms can be restricted in some way so that they can be translated into low-level formalisms, which in turn can be verified again. In particular, the interest of this paper is on high level nets which use the nets-within-nets paradigm, formalisms in which the tokens of a Petri nets can be structured themselves as Petri net. The two formalisms analyzed are *reference nets*, the formalism used as a basis for the Renew tool, and *hypernets*, another nets-within-nets formalism with particular restrictions that allow the expansion toward an equivalent 1-safe Petri nets. In this paper we will show how to translate a subset of the high-level reference net formalism into hypernets, which in turn can be easily translated into 1-safe nets. These can then be analysed by existing toolsets. The main result of our work is the implementation of a Renew hypernet plugin which incorporates features for computing S-invariants, and features for model checking a hypernet. As far as we know, this is the first time that analysis techniques typical of Petri nets has been implemented in a tool which support the nets-within-nets paradigm, and it is mature enough to be used in a real application context. In the rest of the paper when we will talk about invariants we are always referring to S-invariants.

The paper is structured into the following sections. Following this introduction the theoretical and technical background is shortly discussed in section 2. This section will focus on the reference net and hypernet formalisms. In section 3 we will show how to translate reference nets into hypernets and determine the prerequisites for analysis. Section 4 describes the Hypernet plugin created for Renew. Section 5 gives a short example how these different tools are incorporated and used to analyse a given net. The conclusion of the paper is found in section 6.

2 Background and related work

In this section we will introduce by means of examples the basic theoretical formalisms used in this paper, as well as motivate why we have chosen them as our means of verification and modelling. The interested reader can find them in the cited references. In general Petri nets offer a simple way of modelling concurrent behaviour of a system. Higher level nets often introduce abstractions from the simple net models, which offer structures and methods not available to or difficult to model in low-level Petri nets. One major such abstraction is the idea of nets within nets, introduced in [11] for Object Petri nets. This paradigm allows for arbitrary nets to be the tokens of other nets. In this way it is possible to model

the behaviour and interaction of different entities within a complex system, all modelled with Petri nets. Using these formalisms to model and even implement software systems is quite natural. Of course high-level Petri nets and especially formalisms following the nets-within-nets idea are far more complex then the relatively simple low-level Petri nets. This increases the effort and complexity of verifying properties within these nets, which is the main motivation of this paper.

In the following subsections we will describe the reference and hypernet formalisms.

2.1 Reference Nets and Renew

The reference net formalism serves as the starting point of our examinations. It was described in [7]. It is a high level Petri net formalism based on the nets-within-nets paradigm. In this formalism it is possible for tokens within a net to be almost arbitrary objects and especially other Petri nets. Nets can then be used like tokens within their respective so-called system net, but it is also possible to let nets of different layers communicate with one another. The reference net formalism uses reference semantics. This means that tokens within a net do not exclusively correspond to their object/net (value semantics), but only reference their object/net. As a result of this multiple tokens can refer to the same object. This makes it possible to express complex systems in a natural way.

Communication between different net instances within the reference net formalism is handled via synchronous channels, based on the concepts proposed in [5]. Synchronous channels connect two transitions during firing. Transitions inscribed with a synchronous channel can only fire synchronously, meaning that both transitions involved have to be activated before firing can happen. During firing arbitrary objects can be transmitted bidirectionally over the channel. While the exchange of data is bidirectional there is a difference in the handling of the two transitions. The transition, or more accurately the inscription of the transition, initiating the firing is called the *downlink*. The downlink must know the name of the net in which the other transition, the so-called *uplink*, is located. The inscription of the downlink has the form *netname:channelname(parameters)*, in which the parameters are the objects being send and received during firing. If the downlink calls an uplink located in the same net the net name is simply replaced by the keyword *this*. The uplink's inscription is similar, but looses the net name, so that it has the form *:channelname(parameters)*. Uplinks are not exclusive to one downlink and can be called from multiple downlinks, so that this construct can be used in a flexible way. It is also possible to synchronise transitions over different abstraction levels. While during firing one downlink is always linked to just one uplink, it is possible to inscribe one transition with multiple downlinks, so that more than two transitions can fire simultaneously.

Figure 1 shows a simple example of a reference net system. The example was modelled using the Renew tool, which will be described later. It models a producer/consumer system, which holds an arbitrary number of producers and consumers. The system consists of three kinds of nets: the system net, the

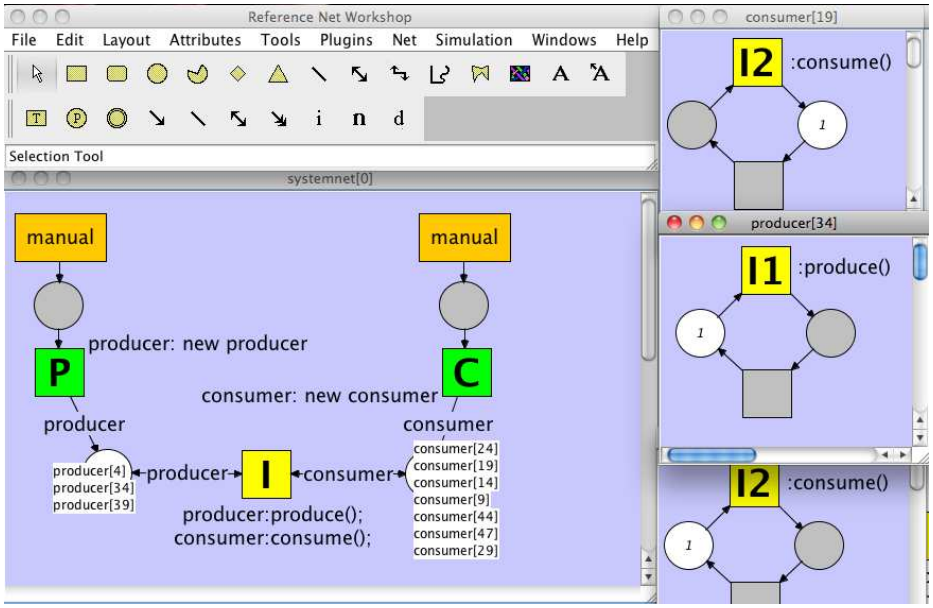


Fig. 1. Reference net example

producer nets and the consumer nets. The producer and consumer nets both possess the same basic structure, but use different channels. The system net serves as a kind of container for the other nets. The transitions labeled **manual** initiate the creation of new producers and consumers by creating new tokens when a user manually fires them during simulation[3]. The transitions labeled C and P actually create new producer or consumer nets when firing. These new nets are put onto the places below the transitions. The transition labeled I synchronises the firing of a transition in one consumer and one producer each (labeled I1 and I2 in the other nets). In this way it is possible to simulate the behaviour in such a way, that whenever a producer produces a product an arbitrary consumer consumes it. It is of course possible to enhance this model by, for example, adding an intermediary storage, which can store items from arbitrary producers until consumers need them. Another way of making the model more realistic is to explicitly model the products as nets as well. That way they would not just be simple tokens but actual objects being exchanged via the synchronous channels between the producers and consumers. In this case the parameters of the channels would be the nets, which would be transmitted from within the producer nets into the consumer nets.

The Petri net editor and simulator Renew (The **RE**ference **NE**t **W**orkshop) was developed alongside the reference net formalism, and is also described in [7] as well as in [8]. It features all the necessary tools needed to create, modify, simu-

[3] This is a special function of the Renew tool, which was used for this example.

late and examine Petri nets of different formalisms. It is predominantly used for reference nets, but can be enhanced and extended to support other formalisms. It is fully plugin based, meaning that all functionality is provided by a number plugins that can be chosen, depending on the specific needs. Plugins can encapsulate tools, like a file navigator or certain predefined net components, or extensions to the standard reference net formalism, like hypernets or workflow nets. RENEW is freely available online and is being further developed and maintained by the Theoretical Foundations Group of the Department for Informatics of the University of Hamburg. Since the tool supports the idea of nets within nets and is flexible enough to support multiple formalisms, it was chosen as the basic environment for the examinations of this paper.

2.2 Hypernets

As we will discuss later in section 3, we introduce hypernets in this paper because they have been used as a restriction of the reference nets formalism to allow property verification in RENEW. Hypernets are a nets-within-nets formalism introduced to model systems of mobile agents [2]. After their introduction several studies has been conducted on hypernets. In [3] it has been shown that it is possible to expand a hypernet in a 1-safe Petri net in such a way that the (hyper) reachability graph of the hypernet is equivalent to the reachability graph of the 1-safe net. In [1] a class of transition system, called agent aware transition systems, has been introduced to describe the behaviour of hypernets. In order to model a class of membrane systems, a generalisation of the hypernet formalism which relaxes some constraints of the basic formalism was introduced under the name of generalised hypernet in [4], and a theorem proving the existence of an expansion towards 1-safe nets for generalised hypernets was proved in [9].

Due to technical limitations in the RENEW tool only the basic version of the formalismi [3] has been implemented. Now we will informally discuss how hypernets work by means of an example. From a structural point of view a hypernet is a collection of (possibly empty) agents $\mathcal{N} = \{A_1, A_2, ..., A_n\}$, which are modelled as particular Petri nets. A state of a hypernet is obtained associating to each one of the A_i agents (nets), but one, a place p belonging to one of the other agents. That place will be the place which contains the agent A_i. This containment relation induces a hierarchical structure which by definition must be a tree. The root of the tree is the only agent which is not associated to any place (this agent is the system net).

The system evolves moving agents from place to place. A peculiar characteristic of hypernets is that the hierarchical structure is not static, but an agent can be moved from a place p belonging to an agent A_i, to a place q belonging to a distinct agent A_j. Another characteristic of hypernets is that agents cannot be created or destroyed. To ensure this "law of conservation of tokens" each net representing an agent is structured as a set of *modules* which have the structure of synchronised state machines, enriched with some *communication* places that allow the exchange of tokens between two agents close in the hierarchy. Agents

and modules have a sort, and an agent can only travel along modules of the same sort.

In Figure 2, and Figure 3 the hypernet modelling a slightly modified version of the *one seater plane* case of study is drawn. This case of study has been introduced in [3], and models an airport in which planes can do basic things like landing, deplaning/boarding passenger, refuelling, and taking off. The changes we made in regards to the number of travellers in the example, the simplification of the safety refuel check and the part of the hypernet which makes sure a plain is empty when it is being refuelled.

To keep the example simple we considered a version with a plane which has only one seat. We choose to illustrate this example to show in an informal way how hypernets works. Moreover, in Section 5 we will show how it is possible to prove some properties of this simple example using the RENEW plugin we developed.

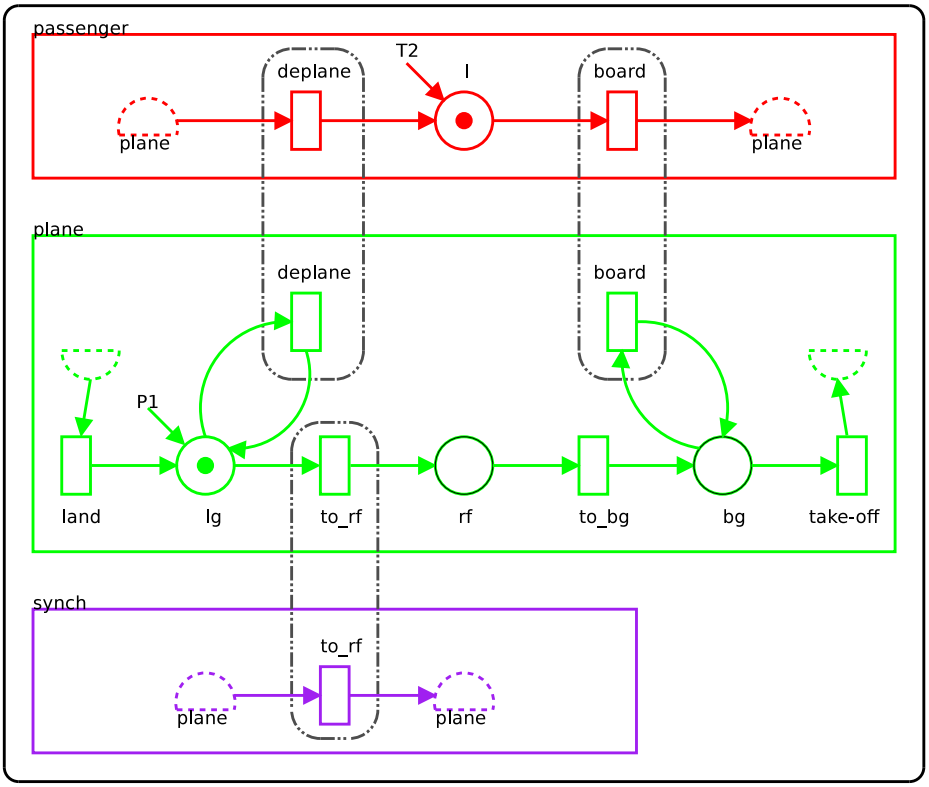


Fig. 2. Airport agent

The agent in Figure 2 models the behaviour of the airport. It has three modules, one for handling passengers, one for handling planes and one for synchroni-

sation purposes. Transition *board* belongs to both module *passenger* and module *plane*, and can only be executed synchronously. The same applies for transitions *deplane* and *to_rf*. Communication places are the dashed half circles. They can either be *up-communication places*, used for communicating with the net at the level immediately above in the hierarchy (such as the two communicating places of the module plane in the airport agent), or *down-communication places*, used to communicate with an agent located in another module of the current net (such as the communication places in the synch, and passenger modules of the airport). In the latter case, a name of a module is provided. In this module there must be an agent ready to provide the *traveling* token which will be moved in the hierarchy, otherwise the transition is not enabled.

For example, transition *deplane* of the passenger module in Figure 2 has an input communication place which indicates that a token is expected. Since this communication place is marked with the *plane* annotation, the traveling token which is being moved to place l must be provided by a plane agent. This plane agent must be located in the input place of transition deplane in module plane of the airport, namely lg. In the example the only agent which can provide a token is $P1$. The traveling token, which must be a passenger, is then selected

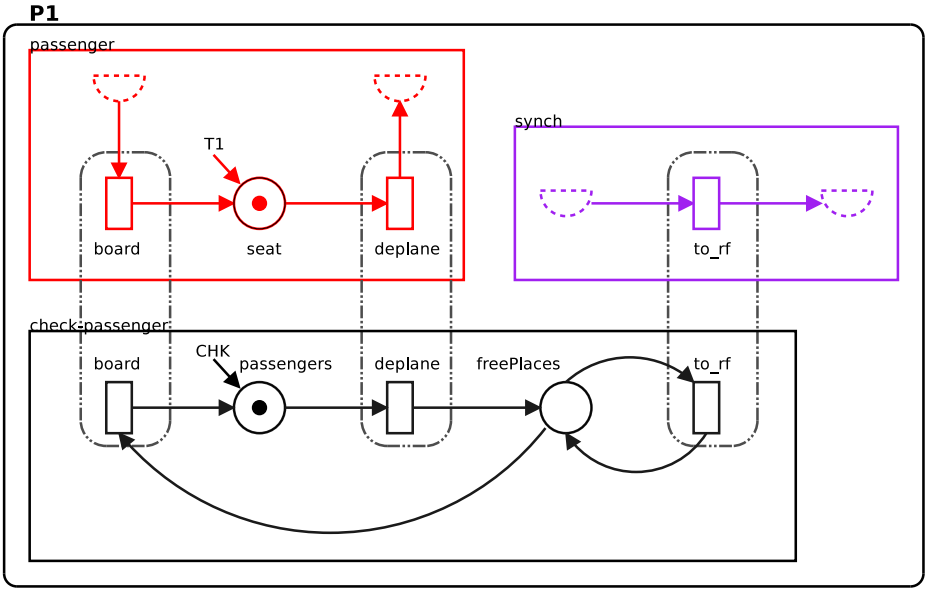


Fig. 3. The $P1$ plane agent shown in Figure 2

and taken from the *seat* place of the plane agent (Figure 3), and moved to l.

Transition *to_rf* is another example of use of communication places. From the airport perspective it is only required that an agent located in the *plane* module has a module *synch* containing with a transition *to_rf* preceeded and

followed by two up-communication places. This requirement is fulfilled by agent $P1$, but from the $P1$ perspective it is also required the enabledness of the synchronized *to_ rf* transition in the module check-passenger. Therefore this configuration *to_ rf* is not enabled because *freePlaces* is not marked.

Hypernet being a high level net model means that the execution of a transition, like *deplane*, has several *firing-modes* [10]. Each firing-mode in a hypernet is a called *consortium*, and is obtained by selecting a transition, a set of agents that contain the transition, and a set of *passive* agents that will be moved as shown in the previous example when the consortium fires. For example, one enabled consortium is the one we just discussed which moves the agent $T1$ from place *seat* of the plane, to place *rf* of the airport agent that we just discussed. Another consortium is corresponding to agent $T2$, which in the configuration shown in the example is not enabled since $T2$ is not located in place *seat*.

One of the most important features of hypernets is that they have a straightforward expansion towards a behaviourally equivalent 1-safe nets. This expansion not only gives hypernets a precise semantics in terms of a well known Petri nets basic model, but also guarantees the possibility to reinterpret on hypernet all the analysis techniques developed for the basic model. The 1-safe net is built in the following way:

- For each agent A, and for each place p in the hypernet a place named $\langle A, p\rangle$ is added in the corresponding 1-safe net. A token in this place means that A is located in p,
- For each consortium Γ in the hypernet a transition named t_Γ is added in the 1-safe net,
- An arc is added from a place $\langle A, p\rangle$, to a transition t_Γ if A is a passive agent in Γ, and p is the input place from which the agent A comes.
- An arc is added from a transition t_Γ, to a place $\langle A, p\rangle$ if A is a passive agent in Γ, and p is the output place where the agent A is going to.

Finally, a place $\langle A, p\rangle$ of the 1-safe net is marked if in the initial configuration of the hypernet agent A is located in place p.

For example, the *one seater plane* case of study we just discussed is translated in the 1-safe net shown in Figure 4. Plane $P1$ can be in places lg, rf, bg in the hypernet, thus the 1-safe net contains places $\langle P1, lg\rangle$, $\langle P1, lg\rangle$,$\langle P1, lg\rangle$. The same must be done for traveler agents, and for the CHK check agent. Since transition *deplane* in the hypernet has two firing-modes, in the 1-safe net two transitions which models each of the firing modes of *deplane* are added (for simplicity both called *deplane*). The same has been done for transition *board*. The firing of a transition in the 1-safe net exactly models what happens when a consortium fires in the hypernet.

As already mentioned, it can be demonstrated that this net is 1-safe, and has a reachability graph isomorphic to the one of the corresponding hypernet. Details, formal definitions, and proofs discussed can be found here for hypernets in [3], and in [9] for the generalization version.

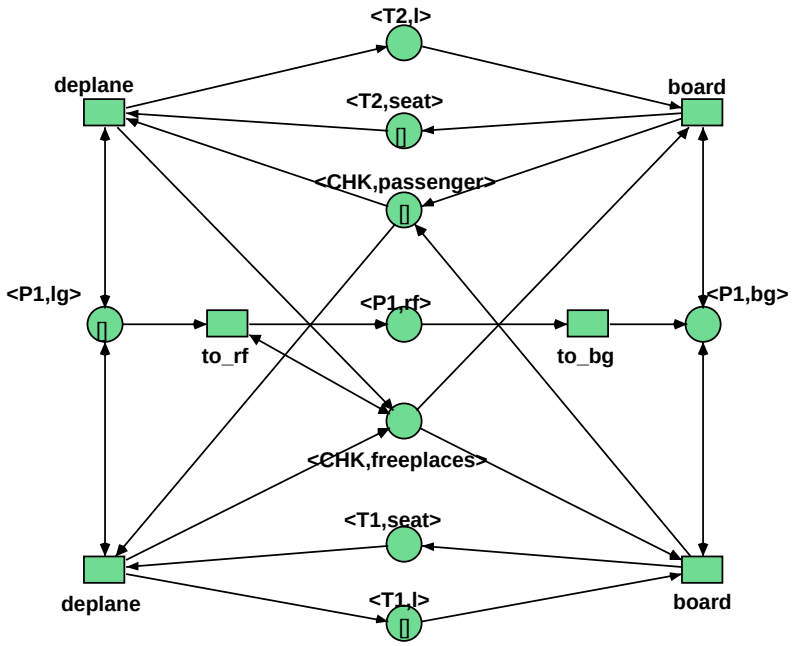


Fig. 4. The expansion toward 1-safe net of the hypernet in Figure 2 and Figure 3

3 Restricting Reference Nets to Hypernet

The main motivation for using high level nets is that, given a system, it is possible to obtain a model of the system with an high level net which is smaller compared to the model obtained using basic Petri nets. However, if you are not careful, the increase of the modelling power decreases the decision power of the model. For example, in [6] it was shown that, even considering a simple subclass of reference nets with one system net, and several references to an object net, the reachability problem becomes undecidable.

It is in this perspective that the implementation of the hypernet formalism as a plugin of the Renew tool has been made. Restricting reference net is probably the most intuitive way to use verification techniques in Renew. In particular, the use of a nets-within-nets formalism like hypernets as a restriction permits the use of the nets-within-nets paradigm, which is probably the most intresting feature in Renew. The original contribute of the paper is to show how this plugin allows the use of verification techniques, like invariants and CTL model checking, to check properties of systems which are suitable to be modeled with the the nets-within-nets paradigm.

4 The Hypernet Plugin

From a technical point of view the implementation of a new formalism in Renew is done using a plugin mechanism. The most important method contained in the classes implementing the plugin is a *compile* method which takes as input a *shadow* net, a set of Java objects containing all the information about the net the user has drawn in the graphical editor of Renew, and transform it in a set of Java objects used by the simulator engine to simulate the net. This compile method is responsible for checking that the net drawn by the user is an actual hypernet in our case. In particular, in order to be able to use Renew as a hypernet simulator, the arc and transition inscriptions used in the modeling process must be restricted in such a way that the drawn net is a hypernet. Therefore the restrictions applied in the plugin are the following:

- Inscriptions (tokens) inside places can only be in the following forms: *identifier* or *identifier:netType*. In the first case the identifier represent the name of an empty net, and will be treated by the simulator engine as an black token; in the second case a new instance of the net *netType* will be created and placed inside the place.
- Inscriptions on arcs are restricted to single variables only. Each arc must contain exactly one variable inscription.
- The inscriptions of input (output) arcs must not be duplicated. In this way it is possible to preserve the identity of nets: duplication of tokens is forbidden.
- Balancing of transition has to be checked, i.e.: the set of variable names used to inscribe input arcs must coincide with the set of variable names used to inscribe output arcs.
- Communication places are deleted, and are simulated by means of synchronous channels. These channels are counted when checking transition balance.

For example, the airport agent shown in Figure 2 can be drawn as a hypernet in Renew using the net shown in Figure 5. The traveler empty tokens are place inscriptions $T1$ and $T2$, and the plane net instance is created by the $P1 : place$ inscription. Each transition is balanced. For example transition *deplane* in the airport has a bidirectional arc labelled pl, and an output arc labelled pa for which there is a correspondant downling, namely $pl : deplane(pa)$. Each communication place is deleted, and it is replaced with a synchronous channel. *Land* and *takeoff* transitions are equipped with two uplink because they were connected to two up-communication places. *Deplane* and *board* transitions contain two downlinks because they were connected to down-communicating places. The module name used to label communicating places is used to retrieve the variable name used in the downlink.

The $P1$ agent of Figure 3 is drawn in the hypernet plugin of Renew with the net in Figure 6. Again, up-communication places are replaced by channels, and transition to_rf must synchronise with the corresponding transition in the airport agent.

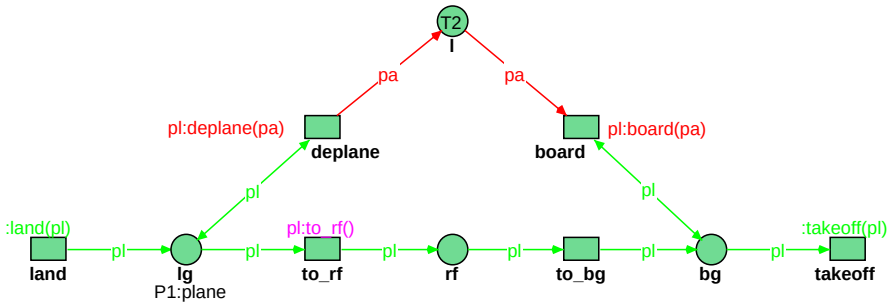


Fig. 5. The airport agent drawn with the hypernet plugin of Renew

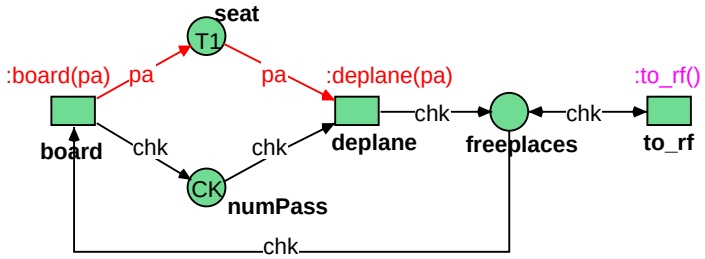


Fig. 6. The plane agent drawn with the hypernet plugin of Renew

As we already mentioned, thanks to the expansion to 1-safe nets it is possible to use verification techniques defined for this class of net to analyse system modelled with a hypernet. Two of the most useful techniques are invariants analysis, and model checking. We explored two possibilities of using them in the plugin we implemented: internal implementation in Renew, or exporting the 1-safe net in a format understandable by other tools. Since implementing these analysis techniques in an efficient way is a difficult task (some tools are very elaborated, and have been implemented over several years), and since very efficient open source tools are available for free, we decided to use external tools to implement invariant analysis, and model checking of a hypernet.

In the following sections we will show how the extensions and incorporation can be used in a practical example.

5 Example

The invariant analysis, and the model checking extensions we implemented in Renew can be used to prove properties of a system. We have chosen the external tools LoLA (see `http://www2.informatik.hu-berlin.de/top/lola/lola.html`) and INA (see `http://www2.informatik.hu-berlin.de/~starke/`

`ina.html`) for analysing purposes. Starting from the hypernet example of Section 2.2, we will prove using invariants that there is never more than one passenger on the plane, and we will prove using the model checker that a plane never refuels if there are a passenger on board.

By running the invariant analysis we get the following invariants:

T2@l	T2@seat	CHK@pass	P1@lg	P1@rf	P1@bg	CHK@freepl	T1@seat	T1@l
0	0	0	1	1	1	0	0	0
1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1
1	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	1	0

The first four invariants are those which guarantee the truth of "law of conservation of agents", achieved thanks to the state machine decomposition in the formalism. For each agent there is a corresponding invariant indicating the places in which that agent can be located. Since the places of each invariant contains only one token in the initial marking, it is mathematically proved that each agent can be only in certain places: the places which are of the same sort of the agent itself. Moreover, these four invariants can also be used to prove that the net is 1-safe: they cover all places of the net, and contain only one token in the initial marking.

The fifth invariant is $\{\langle T2, l\rangle, \langle CHK, numPass\rangle, \langle T1, l\rangle\}$ and contains two tokens in the initial marking. Together with the second and the fourth invariants it can be used to prove that if the place $\langle CHK, numPass\rangle$ is marked then one of the two passenger is seated on the plane. The place is not marked only if both passenger are in the airport.

The sixth invariant is the counterpart of the fifth, and states that only one of the following places can be marked: $\{\langle T2, seat\rangle, \langle CHK, freeplaces\rangle, \langle T1, seat\rangle\}$. The information is clear: only one passenger can be in the *seat* place of the plane. If none of them is in the plane $\langle CHK, freeplaces\rangle$ is marked.

In Figure 7 a screenshot of Renew after the computation of invariants is shown.

While invariants analysis can be launched, and the computed invariants can be analysed to extract information about the system, in order to analyze the system using model checking a formula specified in a temporal logic is needed. Since we choose LoLA, which is a CTL model checker, we need to specify the property we want to verify using this logic. For example, checking the property "if the plane is located in the place representing the refueling station then no passenger is on board" can be done by entering as input of the Renew plugin we implemented the following CTL formula:

$$ALLPATH\ ALWAYS$$
$$NOT\ ((T1.seat = 1\ AND\ P1.rf = 1)\ OR\ (T2.seat = 1\ AND\ P1.rf = 1))$$

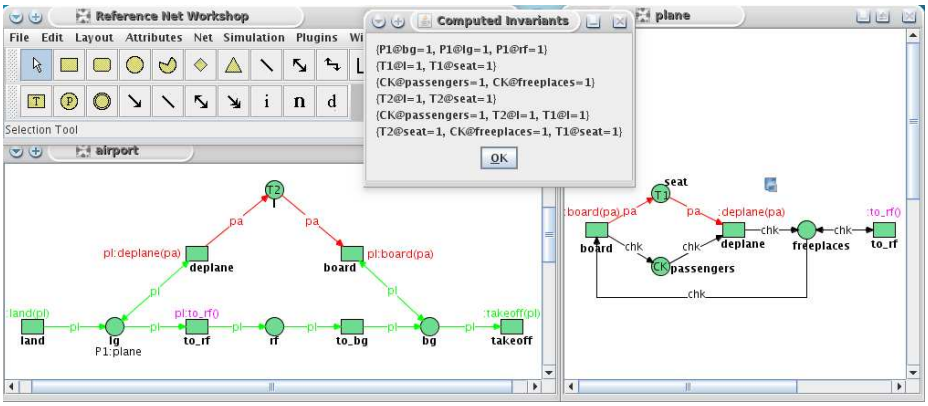


Fig. 7. A screenshot of the invariants computed inside Renew

The formula checks that in every reachable state ($ALLPATH\ ALWAYS$) the situation in which both placed $\langle T1, seat\rangle$ and $\langle P1, rf\rangle$ are marked never occurs (and the same for places $\langle T2, seat\rangle$ and $\langle P1, rf\rangle$). The analysis performed confirms that the truth value of the formula is $true$, which is enough to guarantee that the property is true for the system.

As it can be seen in this simple example, the advantage of using model checking is that it is possible to express, and consequently to verify, more properties compared to invariant analysis. In our example, the information that a plane never refuels if a passenger is on board is not present in the computed invariants, but can be verified using the model checking. However, the drawback is that it is necessary to explore the whole state space of the system in order to verify a property. Invariants are computed on the static structure of the net, which is usually exponentially smaller compared to the state space of the system. In general, in real huge application both the techniques are useful: invariants give a quick overview of some properties of the system, model checking take more time and it can be used to verify specific properties of the system.

6 Conclusion

In this paper we discussed the verification of high-level Petri nets which use the nets within nets paradigm, with particular attention to the reference nets and the hypernets formalisms. We examined them, and we showed how to transform a subset of reference nets into hypernets, which in turn can be transformed into 1-safe nets. We then proceeded to describe the hypernet plugin created for Renew in the course of our work. With the help of this plugin and external tools we can analyse the transformed low-level nets, and in this way verify properties of the high-level net.

The contributions, and the results of this paper are the implementation of a plugin for Renew with which it is possible to draw of a hypernet, to compute

its invariants, and to model check it. With this approach it is now possible to verify properties of systems modelled with net within nets oriented formalisms, such as reference nets and hypernets.

The results of this paper will make it possible to automatically analyse a hypernet, instead of first transforming it by hand, and then analysing the equivalent low-level nets. This will make the verification simpler and more user-friendly, which in turn will make it easier for software engineers to use these techniques in practical use cases. We plan to use these approaches to verify the model of an actually adopted Grid tool for High Energy Physics data analysis, and in the context of the HEROLD project. Future work will also focus on extending the possibilities of the verification, automating the process as far as possible and extending the toolset to other high-level Petri nets formalisms. The flexibility and adaptability of the Renew tool will be a large asset in this endeavour. Finally, the definitions of analysis techniques directly on the high level model, without the need of converting it to a low-level one, is a subject for future investigations, because it will avoid the conversion to low-level nets, which is an expensive operations in term of computational resources.

References

1. M Bednarczyk, L Bernardinello, W Pawłowski, and L Pomello. Modelling and analysing systems of agents by agent-aware transition systems. In F. Fogelman-Soulie, editor, *Mining Massive Data Sets for Security: Advances in Data Mining, Search, Social Networks and Text Mining, and their Applications to Security*, volume 19, pages 103–112. IOS Press, 2008.
2. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. Modelling mobility with Petri Hypernets. In *Recent Trends in Algebraic Development Techniques*, volume 3423/2005 of *Lecture Notes in Computer Science*, pages 28–44. Springer Berlin / Heidelberg, 2005.
3. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. From Petri hypernets to 1-safe nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06, Bericht 272, FBI-HH-B-272/06, 2006*, pages 23–43, June 2006.
4. Luca Bernardinello, Nicola Bonzanni, Marco Mascheroni, and Lucia Pomello. Modeling symport/antiport p systems with a class of hierarchical Petri nets. In *Membrane Computing*, volume Volume 4860/2007 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin / Heidelberg, 2007.
5. Soren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. *Lecture Notes in Computer Science*, 815/1994:159–178, 1994. Application and Theory of Petri Nets 1994.
6. Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 2004.
7. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
8. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In J. Cortadella and W. Reisig, editors,

International Conference on Application and Theory of Petri Nets 2004, volume 3099 of *Lecture Notes in Computer Science*, pages 484 – 493. Springer-Verlag, 2004.

9. Marco Mascheroni. Generalized hypernets and their semantics. In *Proceedings of the Fith International Workshop on Modelling of Objects, Components and Agents, MOCA'09, Bericht 290, 2009*, pages 87–106, September 2009.
10. Einar Smith. Principles of high-level net theory. In *Lectures on Petri Nets I: Basic Models*, volume Volume 1491/1998 of *Lecture Notes in Computer Science*, pages 174–210. Springer Berlin / Heidelberg, 1998.
11. Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advanced Course on Petri Nets 2003*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, 2003.

Improving a Workflow Management System with an Agent Flavour

Daniel Moldt, José Quenum, Christine Reese, and Thomas Wagner

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. This paper discusses an application of software agents to improve workflow management systems, with a practical emphasis on Petri net-based systems. The properties of agent technology will be used to gain advantages within the workflow management systems on both a conceptual and practical level. In this paper we discuss the theoretical background of our work, the conceptual idea and approach and one possible practical implementation. As a central practical means we use reference nets, a high-level Petri net formalism. These nets are used to model both agents and workflows, which results in a clean and natural integration of both technologies.

Keywords: High-level Petri nets, workflow management systems, multi-agent systems, software architecture.

1 Introduction

Workflows and *Workflow management systems* (WFMS) have been very attractive research topics in the last decades [13, 18, 16]. They provide means to further understand, unambiguously specify and analyse business processes within organisations. According to the state of the art, several heterogeneous WFMS can be combined to perform a specific complex task that cuts across various organisations. While such a combination is currently possible in an ad hoc manner, a more systematic approach demands greater care both from the modelling and implementation perspectives. In this research, we set out to address this limitation. This paper discusses the conceptual approach to achieve the overall goal.

Among the rising software paradigms, the concept of *agent* is a preeminent one. In the last decade, agents have been touted as a most appropriate paradigm to support the design and implementation of decentralised and distributed applications/systems, which yield intelligent behaviour and require a great deal of interoperability. A decentralised application implies an application which consists of autonomous entities. Clearly, these are among the properties one seeks while developing an approach for interorganizational workflows. Therefore, agents can

contribute a great deal in our quest for a systematic approach for interorganizational workflows.

Using agents to design and implement distributed applications across a network is not new in itself. However, the autonomy of the various subparts is not well captured in the overall collaboration. As such, approaches of this kind cannot be generalised for the design and implementation of collaborative applications across various organisations. Concepts such as workflows, which render a clear view of business processes within organisations need to come into play.

Introducing agents in WFMS is not counterintuitive. By virtue of being autonomous, sociable and intelligent, human agents and artificial ones share many similarities. Moreover, human agents' operational mode can be viewed as a set of independent yet interoperable entities. From that angle, a human agent can be viewed as an agent system. Finally, since human agents have to coordinate the various tasks they are involved in at a certain point in time, they can be regarded as a WFMS. We take this human analogy, especially its duality, to show that both, agents and workflows, can be joined in a complex system.

As discussed in the foregoing, WFMS can benefit from agents in various regards. In this paper, we discuss seven points where agents help enhance the level of management in interorganizational workflows. These include distribution, autonomy, interoperability and intelligence. In order to make the resulting approach appealing to new technologies, we structure our assumptions and ideas into a reference architecture, which we believe lays the foundation for more specific and advanced architectures to support collaboration within and between organisations. We do not claim that our current implementation is as powerful as the existing commercial tools. However, thanks to the Petri nets formalism, our implementation holds the potential to properly handle concurrency, robustness and resilience in the future.

The contributions discussed in this paper are a clearer articulation of ideas and intuitions presented in [20–22]. More than all these papers, we elaborate on the underpinnings of the conceptual role agents can play in the new approach. Finally, we discuss the implementations.

The remainder of the paper comes as follows. Section 2 describes the technical and theoretical background of our work. Section 3 gives an overview of our overall architecture. Section 4 examines the conceptual view of our approach, while Section 5 discusses one implementation of our approach. Finally, Section 6 draws conclusions and directions for the future.

2 Frameworks & Formalisms

In this research, MASs are designed following Mulan's (**MUL**ti-**A**gent **N**ets) structure [11, 23]. Mulan has been extended with Capa (**C**oncurrent **A**gent **P**latform **A**rchitecture) in order to comply with Fipa's (**F**oundation for **I**ntelligent and **P**hysical **A**gent (see http://fipa.org)) (communication) standards to support concurrent execution [4]. Mulan and Capa describe the various components of a MAS using reference nets, which can be executed using the Re-

NEW (**RE**ference **NE**ts **W**orkshop (see http://www.renew.de)) tool. The reader should thus note that not only do we offer a formal ground to reason about the behaviours of agents, but we also provide an execution environment for MAS.

Inspired from the *nets within nets* concept introduced by Valk [25], MULAN's structure is a four-layer architecture used to describe a MAS. These layers respectively describe the overall MAS, the agent platforms, the agents and their behaviour, the protocols. Every layer within MULAN's reference architecture is modelled using reference nets, a high level Petri net formalism which will be discussed later.

In order to adhere to FIPA's standards, and especially the communication mechanisms, MULAN has been extended with CAPA. In so doing, CAPA agents can easily interact with any type of FIPA compliant agents.

Each of the key concepts in this paper, agent and MAS on the one hand and workflows on the other hand, is represented using a Petri net formalism. Agents and MAS are represented using reference nets, while workflows and WFMSs are represented using workflow nets implemented as a special kind of reference nets. Reference nets have been introduced in [15]. They follow the philosophy of nets within nets. Reference nets use *Java* as an inscription language, manipulate various types of data structures, and like many other types of high-level Petri net formalisms, offer several types of arcs. Finally they use synchronous channels to synchronise with other nets or Java objects. The treatment of Java objects and net instances is transparent, so that both kinds of artefacts can be exchanged arbitrarily. The workflow nets used in our systems are based on principles of workflow nets introduced in [26]. They are implemented as reference nets and make use of a special task transition introduced in [9]. Moreover, in the Petri net community, tool support is a general trend. Therefore, the RENEW tool has been developed to support quick prototyping of systems or parts of systems using the reference net formalism. RENEW provides an editor to specify and draw the nets as well as a simulation engine to test and validate them.

3 Overall architecture

The results we present in this paper are part of a larger ongoing effort. The systems described in the next sections can be classified within the overall architecture described in [19]. The goal of this architecture is to integrate agent and workflow technologies. The architecture consists of five tiers, built on top of each other. Each tier itself is a layered architecture, which combines various aspects of both technologies. Starting from either a pure workflow or agent system, the architecture gradually evolves into a novel integrated unit system, which equally benefits from both original concepts. The main motivation in building this architecture lies in the shortcomings of each individual concept as is discussed in detail in [19] and [28]. In short, agent technology struggles with offering a clear behavioural view of large distributed systems, but can describe the structure of such a system in a natural way. Workflow technology can easily describe the behavioural view of a complex system, but struggles with the structural view.

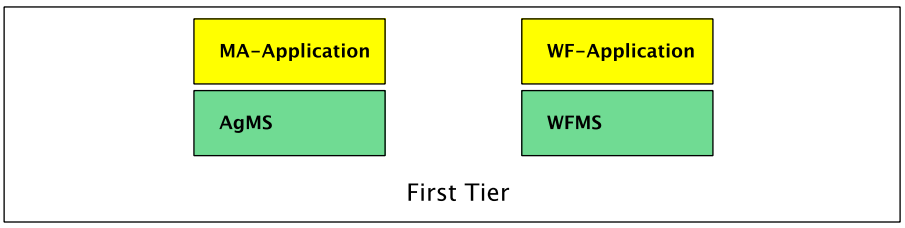


Fig. 1. Architecture of the first tier of the overall architecture, modified from [19]

In combining both technologies we can integrate the views offered by both into one new system which supports both a clean structural and behavioural view. It should be explained that the notion of tiers in this architecture denotes a kind of step-by-step refinement/enhancement on the way to the overall goal of integration. The tiers can be viewed as the layers of the overall *abstract* architecture but they do not correspond to layers within a *concrete* architecture. Each tier modifies the structure of its own layered architecture compared to the previous tier and in doing so enables new or improved aspects to be used. We will now shortly discuss the five tiers of the architecture.

First Tier The first tier is our starting solution to address the limitations of both technologies. It involves either a pure agent management system (AgMS) or a WFMS. Such systems exclusively use workflow or agent technology to provide their functionality. This means that there is almost no integration between the two. Because of this, the architectural view of this tier (see Figure 1) offers only two layers. The bottom depicts the adopted management system (either agent or workflow), on top of which an application lies. Examples of systems, which can be classified into this tier, are Mulan and Capa on the agent side and WFMS like ADEPT (see [3]) or WIFAi (see [24]) on the workflow side.

Second Tier In the second tier, one of the paradigms is used to realise the other. In other words, we use agents to design a WFMS, and vice versa. Because of this, there are two variants of the second tier (agents in the background or workflows in the background). The architectural view of this tier (see Figure 2) offers three layers. The topmost layer still represents an application but between the bottom management system and the application an intermediate layer has been added. This layer implements a management system for the alternative concept using the functionality of the bottom layer. For example, if the bottom layer is an AgMS, then the intermediate layer is a WFMS based on agents. The application layer of this tier uses *only* the concept provided by the intermediate layer.

Building on the first tier the second tier can be achieved by designing the application within the first tier to be the required management system. On top of that management system another application can then be built, turning the application layer of the first tier into the intermediate layer of the second tier.

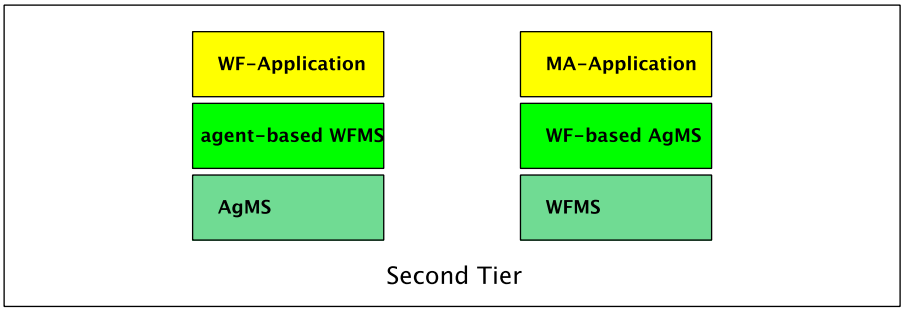


Fig. 2. Architecture of the second tier of the overall architecture, modified from [19]

Compared to the first tier, the advantage is that one can perceive an integration of the concepts. However, the available constructs only affect the background instead of being directly available in the application layer. For example, distribution, interoperability, etc. are facilitated in WFMS using agents.

In the remainder of this paper, the higher level tiers will only be based on the variation using agents in the background, i.e. the one including an AgMS, an agent-based WMFS (AgWFMS) and an application. Examples for these kind of systems are detailed in [6], [7] and [10].

Third Tier The third tier greatly enhances the application development by employing both agents and workflows. This results in an arbitrary degree of integration between agents and workflows. In addition to the interface between the application and intermediate layer, this tier allows direct access from the application layer to the bottom management system. In practice, the application can thus use both the interfaces offered by the core AgMS and the AgWFMS. Consequently, the key functionality of the AgMS is then combined with that of the AgWFMS. The architectural view of this tier (see Figure 3) only adds a direct connection between the application and the bottom layer, compared to the second tier. While this additional connection is a clear advantage over the previous tier by expanding potential and flexibility, it suffers from a major limitation. The resulting system is completely unstructured, i.e., the relation and integration between agent and workflow needs to be potentially re-invented for each application. Consequently it becomes very difficult to harness the power of this tier, especially the efficient design of complex systems. In order to reach a structured integration of both concepts within the architecture, we need to take one step back and limit the immense possibilities offered by this tier.

Fourth Tier The fourth tier adds an integration layer to the architecture, which is responsible for restricting the possibilities of the third tier in order to provide an explicit structure for the application developed on it (see Figure 4). Through this integration both technologies are used in the background and the relationship between agents and workflows is pre-defined. However, only *one* perspective is

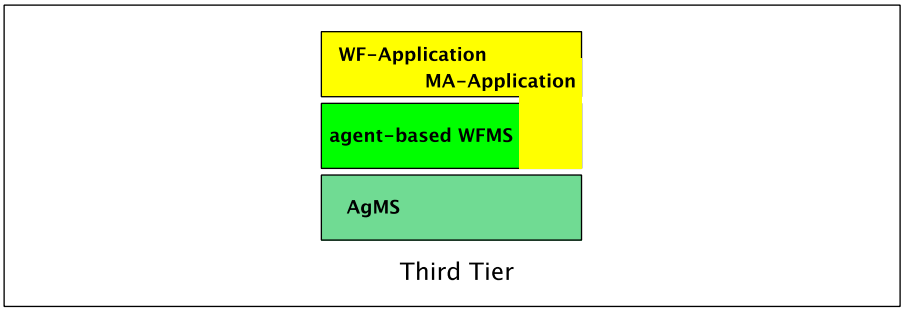


Fig. 3. Architecture of the third tier of the overall architecture, modified from [19]

supported when modelling on the application layer. In this way it provides an abstraction and thus a higher level of modelling.

In order to achieve the desired explicit structure, this tier basically reduces the functionality offered to the application layer compared to the third tier. It refocuses on either agents or workflows as the exclusive main abstraction for application development. There are once again two distinct variations of this tier, one offering agents to the application (called *workflowagents*; left hand side of Figure 4), the other one offering workflows (called *agentworkflows*; right hand side of Figure 4). The integration layer provides exclusively WFMS or AgMS functionality, but uses both technologies in the background. This means that an agent application possesses parts and aspects of workflows and vice versa (in the other variation). In this way the possibilities of this tier are, opposed to the third tier, restricted, because we refocus on just one technology. But by doing this, we gain a much more powerful means of supporting one of the two technologies. The integration of both variations will take place in the fifth tier. For now we need both variations in order to create the desired structure within the relation between agents and workflows in both directions separately.

It is worth noting that, even though we are looking at either workflows or agents at the top layer again, the main difference between this tier and the second tier is that we no longer have one concept realising the other. Rather, we obtain a successful combination of both, i.e., agents and workflows working side by side and benefiting from each other. The agentworkflow variation of this tier is the main focus of this paper and will be discussed in detail in the main sections.

Fifth Tier This tier introduces the concept of *unit*, an abstraction to any entity involved in the design of the system. Units offer both the facets of agents and workflows. In order to achieve this, the AgMS and WFMS have to be integrated and combined. This results in a novel type of management system, a *unit management system* (UMS). The architecture of this tier can be seen in Figure 5. The integration layer of the fourth tier has been split into two parts, of which UMS represents the upper part. Since one can no longer clearly differentiate between

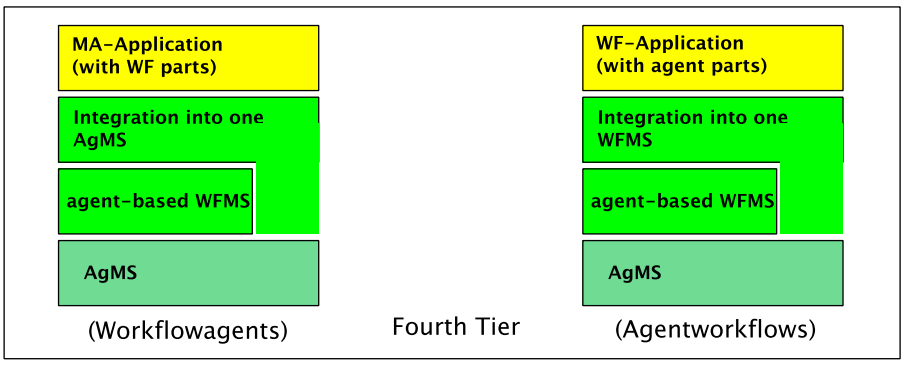


Fig. 4. Architecture of the fourth tier of the overall architecture, modified from [19]

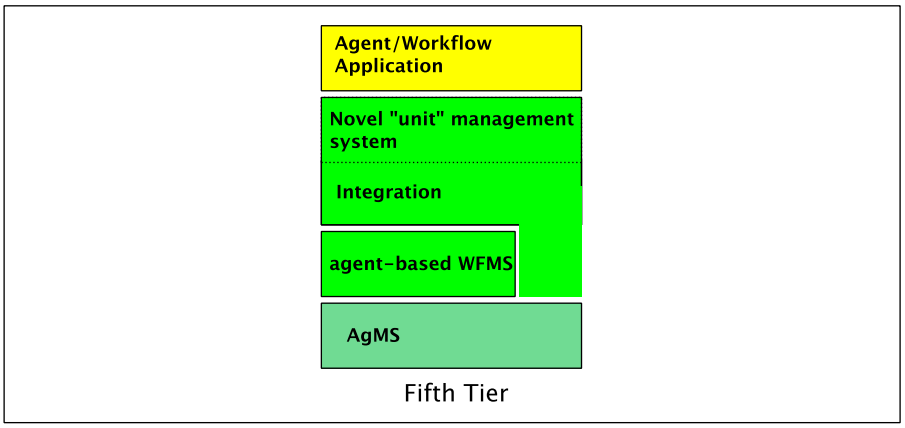


Fig. 5. Architecture of the fifth tier of the overall architecture, modified from [19]

agents and workflows, the application layer is simply called unit application, also referred to as agent/workflow applications in [19]. In merging both agent and workflow concepts into a single unit concept, both the structural view (from the MAS) and the behavioural one (from the workflows) are available. Note that both these views are available during runtime and design time.

4 Conceptual View

In this section we will discuss our conceptual approach to improving workflow management with agent technology. As stated before our goal is to use aspects of software agents to benefit the execution and management of workflow instances. Because we use agents to improve workflows we generally refer to this approach as *agentworkflows*. We reason that common properties of agents, like mobility,

autonomy and proactivity, and especially the encapsulation of workflow instances through agents can greatly benefit workflow execution. Having agents in general handle aspects of workflow management can also help distribute the execution in order to spare less powerful resources.

As mentioned before this particular work is just part of a larger effort to integrate workflow and agent technologies in order to achieve a novel approach that combines the advantages of both technologies. The work presented in this paper focuses on the fourth tier of the overall architecture (see Section 3). We discuss the variation of the fourth tier, in which the integration layer offers a WFMS to the application layer. This WFMS strongly relies on the functionality of the AgMS in the background in order to provide its own functionality. This means that the workflows offered to the application possess some properties gained from the agents. How this can be achieved will be discussed in this section, as well as the advantages and disadvantages of this approach,.

One of the core ideas behind our approach is to encapsulate workflow instances through autonomous software agents. With this it is possible to transfer properties of that software agent directly to the workflow instance. In other words the clear separation between agent and workflow begins to diminish. This is a key concept of the overall architecture and is important for the fifth tier. Another important aspect of our approach is to also consider other entities of the workflow management system as agents. Having the general functionality of the WFMS be provided by agents is already part of an AgWFMS of the second tier. In our approach agents can be used to realise any of the elements (e.g., tasks, users, resources, etc.) of the workflow as well. In [17] for example, we used agents to realise activities. Thanks to the agent technology, the resulting WFMS does not only focus on the behaviour of the system, as was the case in the previous tiers of the overall architecture, it also emphasises the structure of the system. This paper focusses on the aspect of encapsulating workflow instances through agents. Other aspects are considered but will not be discussed in great detail.

We will now examine some of the principal and conceptual areas and aspects in which workflows can benefit from agents. The following points will directly cover some agent properties and their particular benefits but will also include some general observations.

Encapsulation In general the encapsulation of one or more workflow instances through agents can be seen as a prerequisite to opening up many of the possibilities offered by the agent-oriented paradigm. Without this concept it would be hard or impossible to transfer other agent properties over to the workflows. Nonetheless the encapsulation also benefits the workflows in more ways than that. For example the encapsulation provides the workflows with an even clearer identity within the overall system since they can now be identified in the same way as the other elements (agents) of the system. This makes it easier to monitor, observe and analyse the system, which in turn makes maintenance and improvement more efficient. A disadvantage of the encapsulation is that the number of agents active is possibly, drastically increased, depending on how the encapsulation is handled. This may pose

problems on less powerful systems, which simply cannot handle this number of agents or the communication between them. However, since agent architectures are generally built to efficiently handle communication, this should not pose a real problem in practical use.

Mobility By allowing workflows to gain agent mobility they benefit in a variety of possibilities. In the context of software agents mobility describes the capability of a software agent to discontinue its execution within one execution environment (agent platform), migrate to another environment and continue the execution there starting off from its previous state. For agentworkflows this means that the execution of a workflow can be discontinued on one instance of an executing WFMS and continued on another WFMS. Practically this can be used if certain resources needed for the execution of a workflow are not available on every platform. This can include particular (groups of) users or certain, possibly critical data. Another use case for this property is to have a workflow instance migrate not because certain resources are needed, but because its home platform is beginning shutdown or because another platform carries less of a load then the home platform. This use of mobility can lead to improved flexibility, efficiency and fault tolerance.

Autonomy One of the key concepts of the agent paradigm is that agents are autonomous entities. This means that, to a certain degree, they are independent of their environment and can choose for themselves whether to execute an action or not. In the context of agentworkflows this property can be used in a number of ways. It can for example be used as a kind of access control to critical data for which an agent is responsible. This can be a workflow instance but also other entities like activities or the handling of users. Another use of this becomes relevant if combined with mobility. An agent migrating to another platform to access certain data or perform certain actions can do this relatively independent from the other agents and software constructs of that platform, if, of course, it has all the necessary permissions.

Intelligence Intelligence in software agents can be used to describe a multitude of aspects. One major aspect is the ability of certain agents to proactively decide by themselves which actions to take. In the context of workflow management this can be used to predetermine which users should be offered certain tasks, taking variables like workload into consideration. At this point reactivity of agents also comes into play. Software agents can react to events in their environment and adapt according to the situation. For example if there is an error during the execution of a task the agent could observe this and retry the action with changed parameters. Another very interesting aspect where intelligence, proactivity, reactivity and adaptiveness can be used is the adaptivity of workflow instances. Changing workflow instances and even entire workflow definitions according to changed circumstances (current or permanent) improves the flexibility and versatility of a WFMS and can be handled in a natural way using agent intelligence.

Distribution The agent oriented paradigm naturally supports the design of distributed software systems. The main reasons for this are the asynchronous message communication and the autonomy of the individual agents. By re-

lying on agents as the main building blocks of a WFMS it is easy to use these predispositions for the distribution of the system. The communication of different parts of the system can be handled through asynchronous messages, which are flexible and versatile. Extending on this idea opens up even more possibilities of using distribution to the advantage of workflows. Interorganizational workflows can benefit from a distributed WFMS, so that their critical information is not stored in some centralised location.

Interoperability The FIPA communication standards are accepted by many widely-used agent frameworks. Adhering to these standards guarantees interoperability between the different involved software systems, independent of agent architecture or framework. This can be translated into workflow management based on agents as well. In this case different WFMS of different providers can work together, as long as they can process the data structures that are exchanged. This aspect is especially important in the context of interorganizational workflows since it allows some freedom for the choice of the different WFMS in the different companies. But also in general use cases interoperability can be used to an advantage. A FIPA-compliant WFMS can request data from any other FIPA-compliant system, which improves the possibilities of the WFMS. Another aspect which is related to this and distribution, is the openness of the system. Through interoperability and distribution it is possible to create flexible and dynamic open systems to which different WFMS can connect to, complete some tasks, and then disconnect again. Open systems can provide users with functionality that is otherwise difficult to obtain without specialised software solutions.

Structure This point is related to the motivation behind our overall architecture. As described, we reason that workflow systems have trouble adequately describing the structure of the system they are modelling, while focussing on the behaviour. Agent systems on the other hand possess a strong focus on this structure. By joining the two in the ways described in this paper we begin to combine this structural view given by the agents with the behavioural view of the workflows. This is mostly related to the encapsulation aspect discussed above, but contains a more abstract view. By relying on agents one can easily describe the current state of an entity including its current location (in regards to distribution), knowledge and behaviour. By adapting this for workflow instances it can already help provide the structural view needed within a distributed system. If the agentworkflow idea is taken even further and every aspect of the system modelled through agents the structural view becomes even more useful. The location (in regards to distribution) of every resource, user and workflow can be determined and displayed in a way that helps monitoring and maintaining the system.

It should be noted that all these properties only unfold their full potential if used in combination. Every one of these properties and aspects possesses some benefits but together with the others new and improved possibilities can be achieved. For example using mobile agents in a distributed environment of many interoperable agent platforms is more advantageous then forcing the same agent

system onto all involved partners. Equally an autonomous, intelligent agent can decide for itself if a migration is reasonable or not and initiate the action accordingly. Using these properties together also strongly improves interorganizational workflow management and execution. While interoperability and distribution already favour this field, the other properties are also useful, especially in collaboration. For example mobility allows for the transmission of data in a natural way, while encapsulation allows for the clear separation of critical data.

The main disadvantage of our approach is that the realisation and handling of these improved workflows are more complex than handling regular workflows. The reason for this is mainly that the new and improved possibilities will be difficult to harness. It can, if used in the wrong way, affect execution in a negative way or even, in the worst case, prohibit correct execution at all. However, if used correctly and efficiently, they offer clear, distinct advantages to workflow execution in general. They offer novel ways of modelling many parts of workflows and can increase efficiency in use.

After discussing the conceptual view in this section we have shown that our approach offers many advantages, but is difficult to realise and handle. For the realisation part we have chosen technologies based on Petri nets. One problem of the conceptual approach is that different kinds of entities (agents and workflows) have to be combined. By choosing Petri nets as a common basis we can partially circumvent this problem, since it is easier to combine the two kinds of entities when they possess the same basis at the lowest level. On the other hand this choice has the problem of not being widely spread and available. However, certain aspects, like concurrency and displaying behaviour, are very easy to model using Petri nets. In the next section we will discuss one prototypical implementation of our conceptual approach using Petri nets, which already covers some of the properties described in this section.

5 Implementation

In this section we will discuss a prototypical implementation of our agentworkflow approach. As mentioned before we use Petri net based technologies to achieve a common basis for the integration and combination of workflow and agent technologies. In particular we use Mulan and Capa for our agents and workflow nets for our workflow functionality (see Section 2). The starting point of the practical work is an AgWFMS of the second tier of the overall architecture. This AgWFMS has been described in detail in [27]. It relies solely on agents to provide the functionality but does not mix the agent and workflow concepts enough to be considered an agentworkflow system.

Before going into the details of the implementation we will shortly discuss how the different properties observed in the conceptual section can be mapped onto Petri nets. Since discussing these aspects in a reasonable extent would go beyond the scope of this paper and since extensive work on this has already been performed and published we will limit this to referring to other contributions. The mobility aspect has been extensively studied, especially in the context of

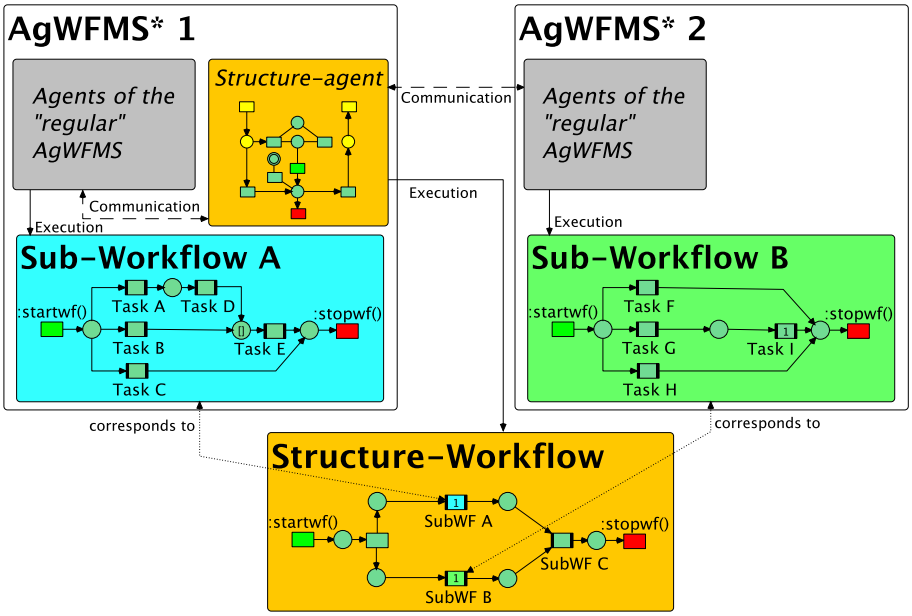


Fig. 6. Principle approach of the S-AgWf

nets within nets, for example in [2] and [12]. The autonomy and intelligence of Petri net agents have been discussed in the context of MULAN in [23]. The encapsulation aspect has been examined for object-oriented nets in [1]. Interoperability and openness have been explored in [14]. The final aspect, the structural view of combining agents and workflows, was discussed in [19].

The practical approach, called structure-agentworkflow (S-AgWf), extends the regular AgWFMS (now called AgWFMS*) to allow for the definition and execution of distributed workflow instances. More precisely, the workflow instances are now hierarchical workflows with nested subprocesses as defined by the WfMC (see [8]).As a consequence the entire system consists of a number of CAPA platforms, which all execute instances of the extended AgWFMS* and are working together. The different AgWFMS* instances are known to one another and messages can be exchanged between them. A more detailed description of the S-AgWf approach can be found in [28].

The basic principle of the S-AgWf can be seen in Figure 6. One agent encapsulates one workflow instance. This agent, called structure-agent, possesses an internal workflow, the structure-workflow. When the structure-agent for a new workflow instance is started, it receives the definition of the structure-workflow from the database agents of the AgWFMS* and instantiates the workflow net. When this initialisation is finished, the execution of the structure-workflow automatically begins. The tasks of the structure-workflow correspond to subworkflows. Sub-workflows can only be executed on certain AgWFMS* instances

within the overall system. The information about which AgWFMS* instance is suitable is stored within the data of the task and can be extracted by the structure-agent. Whenever a task becomes active, the structure-agent assigns itself as the executor of that task. Once this is done the structure-agent queries a special agent of his own platform for a list of all the known AgWFMS* instances currently active. It then compares this list to the information extracted from the task and chooses a suitable AgWFMS* instance. The interface-agent of the chosen instance is then contacted by the structure-agent. The structure-agent asks the interface-agent to instantiate the subworkflow locally and transmits all relevant parameters, like input data etc. The subworkflow is then executed like any other workflow in the regular AgWFMS. Once it has reached the end of its execution the responsible structure-agent is informed and any (optional) results are sent back. The results are transmitted into the structure-workflow net and the structure-agent completes the task, so that the execution can continue. The execution of tasks and subsequent instantiation and execution of sub-workflows is continued, until the end of the structure-workflow is reached. The initiator of the overall workflow is then informed and the structure-agent can terminate.

In the example in Figure 6 two sub-workflows are currently executed on the two different AgWFMS* platforms. The structure-agent responsible for the structure-workflow is communicating with the agents of the two AgWFMS* platforms in order to initiate the execution and receive results. When both sub-workflows are finished the structure-agent will start a final sub-workflow ($SubWF\ C$), before it can conclude the execution of the structure-workflow.

This realisation of the agentworkflow concept offers distinct and practical advantages, but also still suffers from some limitations. The possibility to distribute the execution of workflows is a huge advantage for the otherwise centralised AgWFMS. The support of nested subprocesses allows for interorganizational workflows to be defined and executed. Since the details of the local workflows are not needed globally, the sub-workflows and any critical data they may contain are only known to the local parties. This satisfies the need of interorganizational workflows to secure and conceal confidential and valuable information.

The main limitation of this particular, specialised implementation of the agentworkflow concept is its still centralised nature. If the platform of the structure-agent is disconnected or fails, the entire workflow fails. This could partially be rectified by adding mobility to the structure-agent. It can then easily migrate to another platform, if it discovers any changes in its home platform that might hinder its execution.

The pre-defined relationship between agents and workflows within this system combines the structural aspect of agents with the behavioural aspect of workflows as is the goal in the fourth tier of the overall architecture. The two concepts agent and workflow begin to merge together, since in this system a workflow is an agent and partly vice versa. The practical advantages this particular system gains from this merge mostly consist in a groundwork for further enhancements. Agent autonomy may for example be used to give the structure-agent more control over the workflow instance (e.g. choice over where

sub-workflows are executed), which could result in added flexibility. However the instances within the S-AgWF system already possess certain degrees of distribution (structure-agents communicate with other AgWFMS* platforms in order to execute their structure-workflow), interoperability (the structure-agents exchange FIPA-compliant messages so it is possible to exchange the AgWFMS* platforms with other WFMS if they adhere to the interface) and encapsulation (the structure-workflow is clearly encapsulated by the structure-agent).

6 Conclusion

In this paper, we made a strong case for a systematic introduction of the agent concept to enhance the management of workflows. We pointed out key aspects in which agents improve workflows and their management.

In our quest to develop a systematic approach to support workflow management, we proposed a reference architecture, which builds on an integration of agents and WFMS. The architecture consists of five distinct tiers. These different tiers gradually show how the agent concept first integrates into WFMS and then enhances them on the various aspects we discussed above. This effort culminates in the fifth tier, where both concepts exist alongside each other. As stated in the foregoing, our overall goal in this research is to achieve a seamless integration of agents and WFMS. In this paper, we presented an approach which builds on the agent technology to address WFMS. However, the other variant of the fourth tier, the workflowagents, needs to be considered as well. In it, the main abstraction of the application layer are agents, which strongly rely on the functionality of the WFMS in the background to address the inherent limitations to an agent-based system. Clearly, following that perspective, a WMFS could for example bring its systematic and proof-driven approach to complex task execution. In the future, we wish to explore that perspective as well.

From the lessons learned from both approaches, we expect to collect the amount of information that enables us to design a full-fledged conceptual approach which offers the best of both agent and workflow technologies in one single system, i.e. the fifth tier. Such an approach will balance out the weaknesses of each technology. With the support of high-level Petri nets as a foundational formalism, we are guaranteed of combining structure and behaviour in one representation.

References

1. Ulrich Becker and Daniel Moldt. Objekt-orientierte Konzepte für gefärbte Petrinetze. In Gert Scheschonk and Wolfgang Reisig, editors, *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, Informatik Aktuell, pages 140–151, Berlin Heidelberg New York, 1993. Gesellschaft für Informatik, Springer-Verlag.
2. Lawrence Cabac, Daniel Moldt, Matthias Wester-Ebbinghaus, and Eva Müller. Visual Representation of Mobile Agents – Modeling Mobility within the Prototype MAPA. In Duvigneau and Moldt [5], pages 7–28.

3. Peter Dadam, Manfred Reichert, Stefanie Rinderle-Ma, Kevin Göser, Ulrich Kreher, and Martin Jurisch. Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen. *EMISA Forum*, 29(1):9–28, 2009.
4. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
5. Michael Duvigneau and Daniel Moldt, editors. *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*, number FBI-HH-B-290/09 in Bericht. University of Hamburg, September 2009.
6. Lars Ehrler, Martin Fleurke, Maryam Purvis, and Bastin Tony Roy Savarimuthu. Agent-based workflow management systems (WfMSs) - JBees: a distributed and adaptive WfMS with monitoring and controlling capabilities. *Information Systems and E-Business Management*, 4, Number 1 / January, 2006:5–23, 2005.
7. Andrea Freßmann, Rainer Maximini, and Thomas Sauer. Towards Collaborative Agent-Based Knowledge Support for Time-Critical and Business-Critical Processes. In *Professional Knowledge Management*, volume 3782, pages 420–430, Berlin Heidelberg New York, 2005. Springer-Verlag.
8. David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition. Verfügbar auf http://www.wfmc.org/reference-model.html.
9. Thomas Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 2002.
10. N. R. Jennings, T.J. Norman, and P. Faratin. ADEPT: An Agent-Based Approach to Business Process Management. *ACM SIGMOD Record*, 27:32–39, 1998.
11. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the Structure and Behaviour of Petri Net Agents. In J.M. Colom and M. Koutny, editors, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 224–241. Springer-Verlag, 2001.
12. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling mobility and mobile agents using nets within nets. In Wil van der Aalst and Eike Best, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets 2003 (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 121–139. Springer-Verlag, 2003.
13. Michael Köhler-Bußmeier. Hornets: Nets within Nets combined with Net Algebra. In Karsten Wolf and Giuliana Franceschinis, editors, *International Conference on Application and Theory of Petri Nets (ICATPN'2009)*, volume 5606 of *Lecture Notes in Computer Science*, pages 243–262. Springer-Verlag, 2009.
14. Michael Köhler-Bußmeier. SONAR: Eine sozialtheoretisch fundierte Multiagentensystemarchitektur. In Rolf v. Lüde, Daniel Moldt, and Rüdiger Valk, editors, *Selbstorganisation und Governance in künstlichen und sozialen Systemen*, volume 5 of *Reihe: Wirtschaft – Arbeit – Technik*, chapter 8–12. Lit-Verlag, Münster - Hamburg - London, 2009.
15. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
16. Kolja Markwardt, Daniel Moldt, and Christine Reese. Support of Distributed Software Development by an Agent-based Process Infrastructure. In *MSVVEIS 2008*, 2008.

17. Kolja Markwardt, Daniel Moldt, and Thomas Wagner. Net Agents for Activity Handling in a WFMS. In Thomas Freytag and Andreas Eckleder, editors, *16th German Workshop on Algorithms and Tools for Petri Nets, AWPN 2009, Karlsruhe, Germany, Proceedings*, CEUR Workshop Proceedings, 2009.
18. Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen*. Dissertation, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, August 1996.
19. Christine Reese. *Prozess-Infrastruktur für Agentenanwendungen*. Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2009. Pdf: http://www.sub.uni-hamburg.de/opus/volltexte/2010/4497/.
20. Christine Reese, Jan Ortmann, Daniel Moldt, Sven Offermann, Kolja Lehmann, and Timo Carl. Architecture for Distributed Agent-Based Workflows. In Brian Henderson-Sellers and Michael Winikoff, editors, *Proceedings of the Seventh International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2005), Utrecht, Niederlande, as part of AAMAS 2005 (Autonomous Agents and Multi Agent Systems), July 2005*, pages 42–49, 2005.
21. Christine Reese, Matthias Wester-Ebbinghaus, Till Dörges, Lawrence Cabac, and Daniel Moldt. A Process Infrastructure for Agent Systems. In Mehdi Dastani, Amal El Fallah, Joao Leite, and Paolo Torroni, editors, *MALLOW'007 Proceedings. Workshop LADS'007 Languages, Methodologies and Development Tools for Multi-Agent Systems (LADS)*, pages 97–111, 2007.
22. Christine Reese, Matthias Wester-Ebbinghaus, Till Dörges, Lawrence Cabac, and Daniel Moldt. Introducing a Process Infrastructure for Agent Systems. In Mehdi Dastani, Amal El Fallah, João Leite, and Paolo Torroni, editors, *LADS'007 Languages, Methodologies and Development Tools for Multi-Agent Systems*, volume 5118 of *Lecture Notes in Artificial Intelligence*, pages 225–242, 2008. Revised Selected and Invited Papers.
23. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
24. Michael Tarullo, Daniela Rosca, Jiacun Wang, and William Tepfenhart. WIFAi - A Tool Suite for the Modeling and Enactment of Inter-organizational Workflows. In *SOLI '09. IEEE/INFORMS International Conference on Service Operations, Logistics and Informatics 2009*, pages 764–769. IEEE, 2009.
25. Rüdiger Valk. Concurrency in Communicating Object Petri Nets. In *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 164–195. Springer-Verlag, Berlin Heidelberg New York, 2001.
26. Wil M.P. van der Aalst. Verification of Workflow Nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248, pages 407–426, Berlin Heidelberg New York, 1997. Springer-Verlag.
27. Thomas Wagner. A Centralized Petri Net- and Agent-based Workflow Management System. In Duvigneau and Moldt [5], pages 29–44.
28. Thomas Wagner. Prototypische Realisierung einer Integration von Agenten und Workflows. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2009.

Part III

Short Presentations

IRS-MT: Tool for Intelligent Resource Allocation.

Piotr Chrząstowski-Wachtel[1,2] and Jakub Rauch[1]

[1] Institute of Informatics, Warsaw University,
Banacha 2, PL 02-097 Warszawa, Poland
[2] Warsaw School of Social Sciences and Humanities,
Chodakowska 18/31, PL 03-815 Warszawa, Poland
pch@mimuw.edu.pl, jakub.rauch@gmail.com

Abstract. A tool for optimizing cost and time of a workflow execution with respect to allocation of multi-purpose resources is presented. The optimization is done as a result of simulations, which take into account the cost and time associated with each of the resources, when allocated to transitions in Petri nets representing a workflow. These attributes can be collected and updated based on the logs of the finished instances. The program suggests the best allocation procedures, giving the estimates of the performance of the whole run for all possible decisions.

1 Introduction

Modeling processes as workflows has become quite popular and proved its usefulness in practice. One of the problems associated with running a workflow is the resource management. By resources we mean all components required to run an activity. In our case, their necessity is described by certain requirements (e.g. skills or functions) associated with activities (transitions). With each resource we associate a bunch of skills, so we can choose, which of the resources are to be attached to certain activity at the workflow simulation run-time. The Petri net approach requires collecting all the resources necessary for a given transition before triggering an acitivity to run. One cannot reserve a resource, and keep it busy, while waiting for other resources necessary to run an activity. Only when all resources are available we can make a decision to run (fire) an activity.

It often happens that a resource is requested by different activities. A resource conflict occurs, when a single resource is shared by two enabled transitions. We must make a decision, which activity will use the resource first. We assume here that the resources are re-usable, and that after finshing a transition the resource can be used by another activity.

When we make a decision about the resource allocation, we should take into account several aspects. Usually we try to optimize some quality function, like time or cost of the workflow run. We assume here that during a workflow run we can perform several actions concurrently. Since some of them will be competing for resources, a proper allocation can improve the quality of workflow

run. Engaging proper resources can diminish for instance the delays caused by lack of the only resource requested by a concurrent action and hence waiting for this resource to be released.

As described in [BPS09], there are many essential aspects of resources, which should be taken under consideration during workflow simulation. One of them is already mentioned: the multifunctionality of resources. Resources have attributes describing their skills. In other words these are the abilities to perform certain actions. Each resource is associated with the set of activities, in which it can be used. Other attributes taken under consideration are performance and cost associated with engaging the resource. So we know how fast a resource can do an activity and how much it costs to use it. By expense of a resource we mean here a value per time unit associated with involvement of the resource. A chief accountant probably has a driving license, but using him to drive the documents somewhere can be a waste of his time and precious skills. His cost per hour is much higher than that of a professional driver.

Since workflows can be quite complex, the authors do not see any analitycal approach, which would be effective in the optimization of runs. It is hard to predict all the possible outcomes of the allocations decisions, when resources are in many conflicts. Instead, we propose an experimental approach, which involves the simulation of many runs, taking into account different allocations and estimating the desired measures as a result of allocation decisions. In our prototype IRS-MT (Intelligent Resource Sharing-Modelling Tool) the manager can edit a structured workflow net and set the number of experiments. The program makes random allocations reporting times and costs of the runs. Based on this knowledge the manager can use the suggestions of the program and get a picture of possible outcomes of the decisions taken. The decisions can be made incrementally. After each decision the workflow run advances its state, and when we come to the next decision, a separate simulation will be made, basing on the actual state of the system.

2 Basic definitions

In our tool we consider resources (S), roles (R), requirements ($\mathbb{Q}$) and activities (A). Resources and roles are finite and defined a priori (by designer). For each resource we define a set of skills (roles), which it can use. This is denoted by $F_{SR} : S \to P(R)$ function. Additionally, for each role in a resource, its efficiency may differ. For every resource $s \in S$, we define the *efficiency* function $E_s : F_{SR}(s) \to \mathbb{R}_+$. This function describes how fast a resource performs each of its roles. The lower this value is, the higher is the efficiency (one is the base value). Additionaly, for every resource we define its use cost per time unit as a function $C : S \to \mathbb{N}$. On the other hand, for each activity in a workflow, a set of requirements needed to fire it, should be determined. Every requirement $Q \in \mathbb{Q}$ is a subset of roles. We define a function $F_{AQ}(a)$ which is a bag of requirements associated with the activity. It is important, that every activity has an expected duration time defined and its standard deviation value. Later, in the generation

and simulation phase, only the available resources fulfilling these requirements will be considered for taking part in such activity. And so, $s \in S$ is *fulfilling* $q = \langle R_q \rangle$, iff $R_q \subseteq F_{SR}(s)$. It is important, that if a resource is involved in some activity, it cannot be used by any other activity (there is one exception, which will be covered later in this section).

Initially all the resources are free, available in a pool of idle resources. We assume, that all the resources are reusable, so at the beginning of activity execution the needed resources are collected and at the end all the freed resources will be returned to the pool of idle resources and will be available for further use. The graphical representation of relations between the introduced notions is depicted on Fig 1.

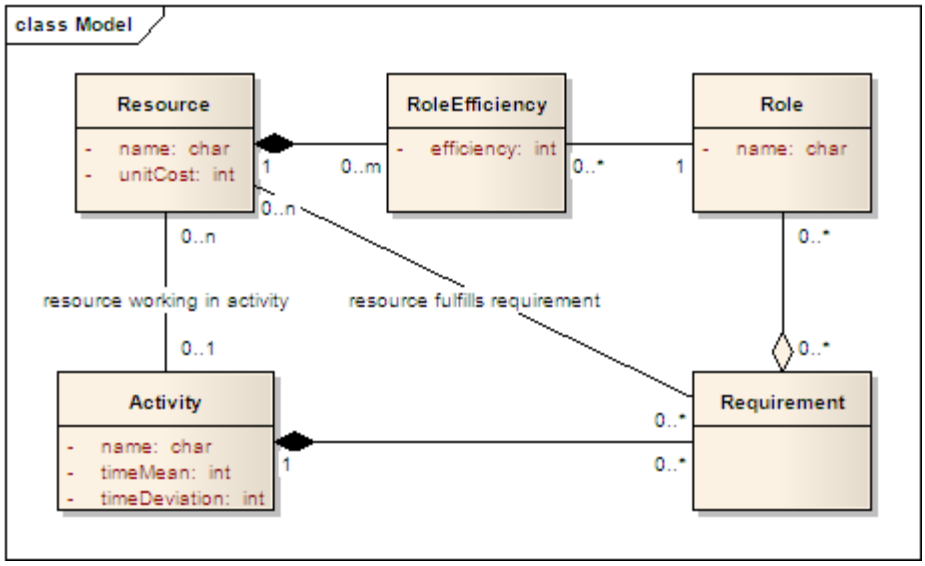


Fig. 1. Model

Our workflows are Petri nets created using five basic refinement patterns, proposed in [PCh03]: sequential-split of a place or transition, parallel-split of a place, choice-split of a transition and loop-split which attaches the spawned transition with a self-loop to a place. The additional rule for attaching a resource place is dual to the loop-split. It just glues a freshly created (resource) place to a transition by a self-loop. From the Petri net perspective we assume here that such place will contain initially a token for each physical resource available. We call the places created by such splits *resource places* or *activity places*. One cannot refine resource places. Even if the transition is refined, the resource will be allocated at the beginning of its execution and released, when it is done. Moreover, every resource place will carry the set of requirements defined for corresponding

activity. If we match the resource places with activities accordingly, we will see, that the requirement assignments are defining the F_{AQ} function.

When we use activity refinement on a transition, which is inside some other activity, we will create a *nested activity*. By its *ancestor activity* we call every other activity, in which this one is nested. Such nested activities do not differ from any other activities, except for the fact, that these can use resources from its ancestors, as long, as the resource will not have one of its roles assigned to requirements in two different activities.

3 Tool overview

The main goal of this work is to provide a a tool, which can be helpful in improving the resource management. We concentrate on optimization of the workflow execution by providing the user with relevant statistical information and letting him make decisions on resource-to-activity assignments. To achieve this, a Petri net defining a workflow with activities and resources, will be created. For this purpose we introduce a *Workflow Designer*. It is the editor for building workflows using refinement patterns. In the editor we create activities, declare the set of requirements and approximate execution duration. On the other hand, we have *Resources Editor*. We use it for defining a pool of resources, assigning roles to them, and determining their effectiveness in each role. The set of defined roles can be modified by an always-visible editor *Roles Viewer*. All these three editors form the *static part* of the tool. Its more detailed description is presented in section 3.1.

After defining the model in the *static part*, we can proceed to the *dynamic part* of the tool. We will use *Generator*, to create sufficent number of random runs. When this is done, the *Simulator*, the *Report Viewer* and the *Bucket Editor* shows up. The first one displays a copy of our workflow, where resource places contain both the requirements and lists of currently available resources, fulfilling given requirements. Here, the tool provides us with information about expected time and cost of workflow completion for each of resource-to-requirement assignment we see. Basing on this knowledge we can decide, which resource should be assigned for current requirement. Every choice causes the expected values to be recalculated, so that we can run the workflow deciding, how resources should be used in the activities. The Report View presents detailed information about expected time and cost of workflow completion. It is synchronized with current simulator state, so all the values are always up-to-date. The Bucket Editor is a tool for storing, and presenting details of the runs, which were manually performed by user in the Simulator.

For example, let us consider the following, simple case. There is a package, which must be delivered to the destination within an hour. We know, that standard travel time by a scooter in current traffic would take around fifty minutes. We have two available scooter drivers: Evan and Gregory. Both of them can do the delivery, but Gregory has got his license for much longer time than Evan, and he shortens the expected delivery time by around 20 percent. Evan, on the

other hand, is still afraid of driving fast and using tricky shortcuts, so his deliveries often take 20 percent longer than normal. However, Gregory's earnings are twice the earnings of Evan. This is the classic case, where no optimal resolution to the problem exists. It must be up to user's decision, whether time or cost is more important, and it is up to our tool to provide the user with information about expected cost and time consequences of each decision. We achieve it by simulating many thousands of runs and preparing the estimates with all aspects of the workflow taken into consideration, i.e.: activities order and dependencies, resources usage conflicts, possible time/cost variations.

3.1 Static Part

To present the tool in more details, we introduce other, more complex example. Let us suppose, that we have two rooms: A and B. We need to plaster and paint both of them. Additionaly, room A needs to be decorated. Here we assume, that a room cannot be painted, unless it is plastered and it cannot be decorated, unless it is painted. Expected time units, required to complete these tasks for each room are following:

- Room A — plastering: 20, painting: 10, decorating: 20;
- Room B — plastering: 10, painting: 20.

This process is presented on Fig. 2. We also need two resources, applicable for the work: *Steven* and *Tom*. *Steven* is an experienced *painter*, with a skill of *plastering*. *Tom*, on the other hand, is a *decorator*, who also can *paint*, but it takes him some more time that it does for *Steven*.

The purpose of the static part of the tool is to model this situation, so that it can be then simulated and later analysed in the dynamic part. We will now explain, how it can be achieved using available functionalities.

Resources Editor Defining resources consists of identifying the available set of people, machines and tools. For each of them, we can define a set of attributes like: use cost per time unit, collection of applicable roles (skills) and the effectiveness in each role. Every resource may have many skills, and many resources can have the same role. As it turns out, we often miss such knowledge during resources allocation planning and we do not take all the benefits from what a resource is capable of doing. Because of that, we can also miss the optimal resolution for given situation. Therefore we need to integrate all these aspects in the analysed context, so that we can consider consequences of particular situations with respect to most important factors. It is worth noticing, that in most popular Human Workflow management tools like Tibco [BS07] or Corel iGraphix, as well as in some academical tools like Yasper [YA06], during the workflow design and simulation phase, the roles are treated as resources. No resource, can have two skills. This, for modelling purposes, is a major limitation. It means, that one resource will never be requested for two activities with different required roles, which tightly limits analysed possibilities. In our tool there are no such boundaries.

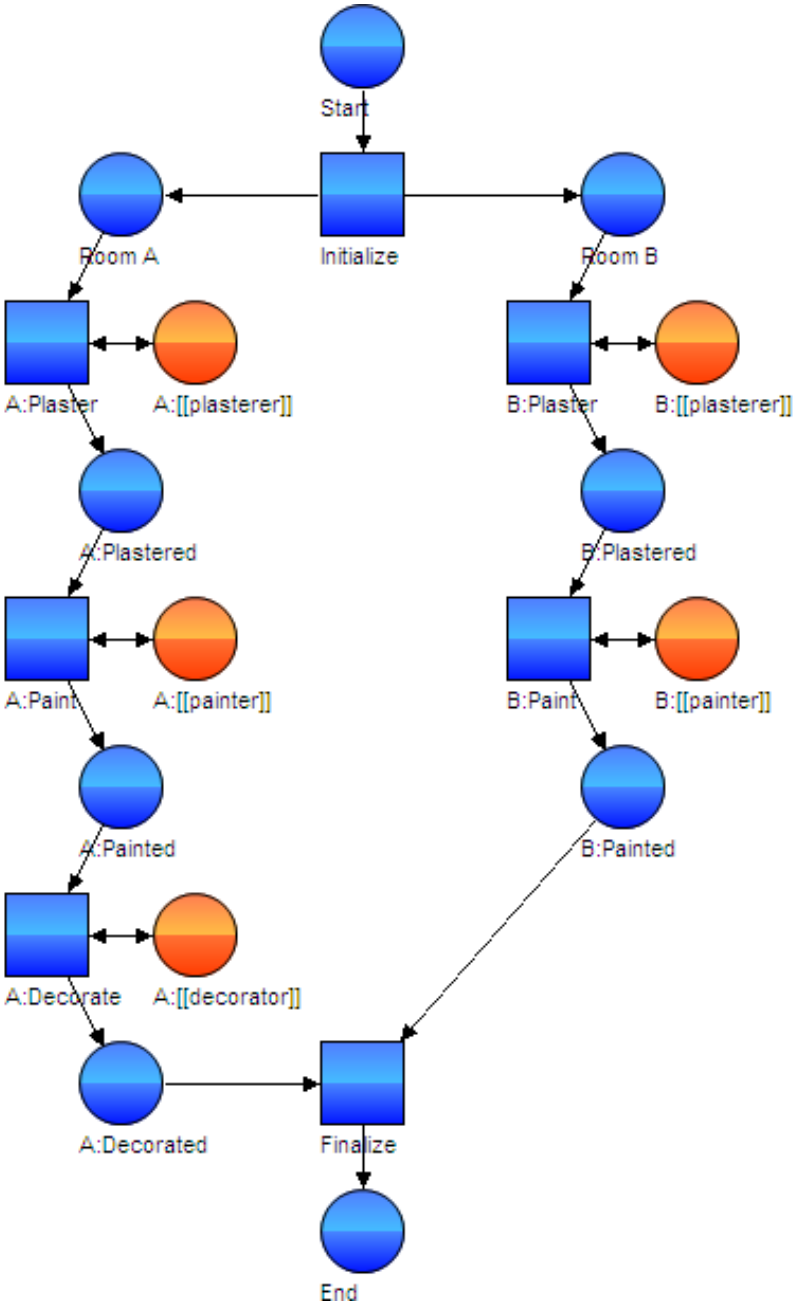


Fig. 2. Sample net

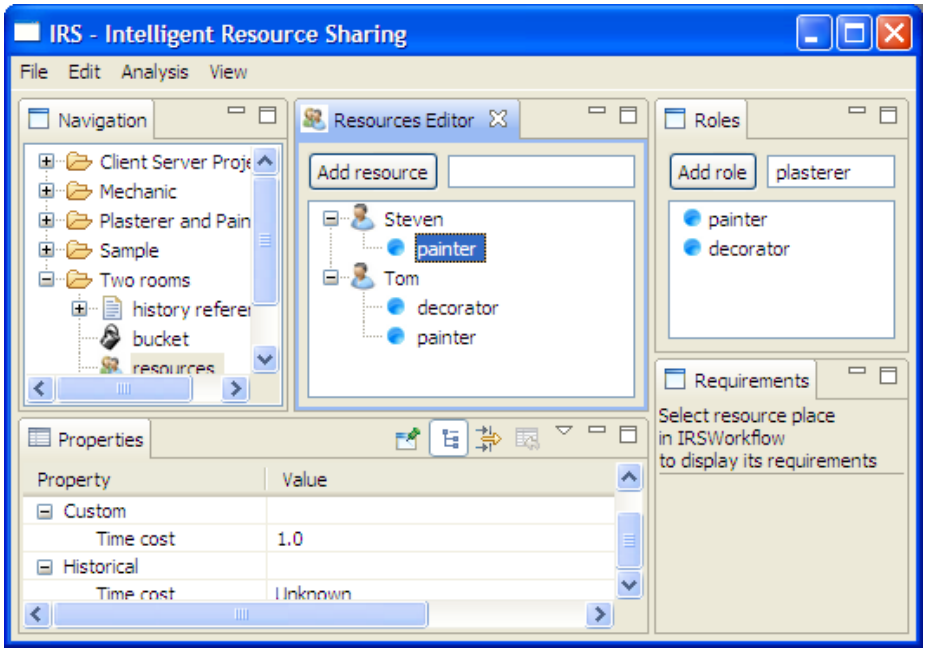


Fig. 3. Resources edition

Fig. 3 presents the pair of our resources: *Steven* and *Tom*. The interface is straightforward and it uses drag and drop features across almost every view. At the depicted state of modeling Steven lacks the plastering skill yet. In order to add it, we type *plasterer* in *Roles View*, press *Add role* button, drag the newly created role from this view and drop it over the *Steven* entry. New skill would get all of the necessary attributes initialized to default values. At the bottom of the window, we can see the *Properties* tab, which, as a context panel, allows us to modify attributes of currently selected object. The information about Steven's efficiency as a *painter* (*Time Cost* row) is displayed here. To be consistent with the descritpion, we should change this value from 1.0 to 0.8. This indicates that activity performed by Steven in a role of *painter* could take 20 percent shorter than normal. We repeat similar scenario for Tom, associating with him the role of painter and decorator and setting the efficiency coefficients for each of these roles. Expenses for each person should be defined also here.

Workflow Designer In the presented tool we can model sound workflows using refinement patterns presented in [PCh03]. To make this process easier it is possible to apply these patterns to any node the refinement tree, including the inner ones. Each node can also have its subtree truncated. The structural approach has been chosen to simplify both the design process and the inner application processing engine. In this tool we introduced the basic set of six refinements.

These might be extended in future by other patterns like communication or synchronization patterns described in [PCh03].

Editor offers additional operations for collapsing and expanding nodes, according to the information held in the refinement tree, so we can view our net at desired level of detail. The application displays all the nodes automatically in the viewer, but we can also move them around manually. The process from the example, has been created using refinement patterns, as shown on Fig. 4. Currently the *B:[[plasterer]]* resource place is selected, so that we can see, which requirements are defined for this requirement place in the bottom right tab *Requirements* (in our case all resource places will be labelled ⟨room name⟩:[[⟨required role⟩]]). As it was mentioned earlier, resource places are created by the *ACTIVITY* pattern, and cannot be further refined. Each resource place is associated with exactly one activity, so it is the right place to hold all the needed requirements for activity.

The whole net has been built using only the *SEQUENCE*, *CONCURRENCY* and *ACTIVITY* patterns. On the Fig. 5, all possible refinement operations are presented both for each standard place (not resource place) and each transition.

Every transition has got a special property, a measure, which describes its *desire to be executed*. We can modify this value to indicate transitions, which should have higher/lower probability of being executed, when in conflict with some other ones. This value is used only when at least two transitions are in conflict. By default this value is set to 1, but if we wish to make some transition to be executed more often, this value should be set accordingly. For instance if we have two active transitions, one with this value set to 3, and the second one to 1, then the first of them will be fired with 75% chance. This way we can declare, which actions are more probable or what kind of situations occur more often.

3.2 Dynamic Part

Statistics Generation Generator is a tool for preparing a list of complete runs with respect to guidelines defined in a workflow project and resources set. For this purpose, a special, extended copy of the designed workflow is created. Apart from copies of all the elements created by the designer, there are additional resource places for automatic resource availability management. It is guaranteed, that before each run, the whole net will be reset. Allocation decisions taken between two different runs are then mutually independent. It can happen that two identical runs could be generated by chance. Each run is executed and recorded using the following algorithm.

Initially, the *in* place of a workflow, is marked by a token, as well as there are tokens in the resource places. In a loop, a complete set of active transitions (activities) is constructed. If the set is empty, and the workflow is finished (a token is present on *out* place of the workflow), then the run is stored, the token from *out* place is moved to back *in* place. If there are at least two enabled transitions, one of them is randomly chosen (according to its *desire to be executed*). The selected enabled transition is fired and the loop is repeated. Note, that some

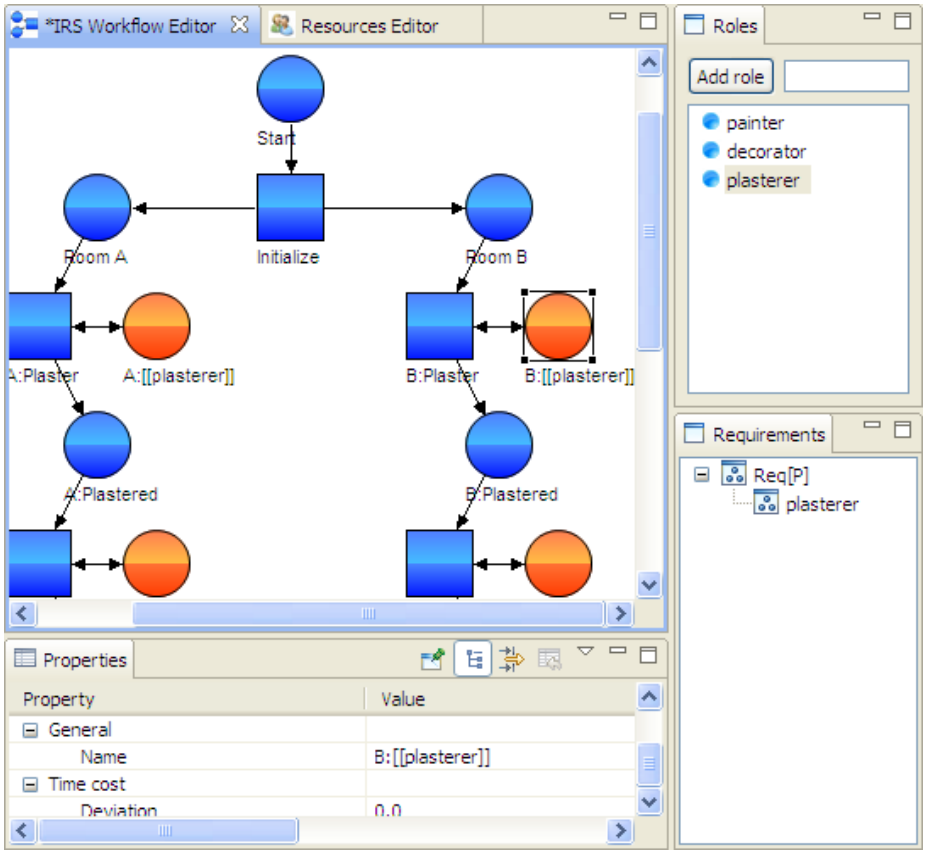


Fig. 4. Designer

of these transitions will indicate the begin or end of some activity. In such case an additional resource acquistion or return will be performed.

When the generator attempts to start an activity, all the requirements are being covered by skills owned by resources assigned to it. When we start one activity, some other can also start or end, so the tool can model various concurrent situations. When an activity ends, assigned resources are freed, the generator updates the workflow timer, the amount of time and cost counters for later analysis. A series of runs allows us to consider usefulness of certain choices in the context of further possible events.

The working time of the generator is dependent mostly on the number of allocations and releases of resources, which corresponds to one run. The computation power is also very important. In our case the generation of 10000 (ten thousand) runs on a standard computer takes no more than a few seconds.

Fig. 5. Refinement patterns

Simulator Taking into account possible allocation decisions, we analyse time and cost of a workflow run. A few similar simulation tools have been reviewed in [BPM05]. On that basis some of the functionalities have been adapted to this tool, and a few flaws have been evaded. As a result, the application provides the simulator of the given net, and gives us a browser of performed runs.

In the report viewer (described later), we can browse all the runs created during generation. All these runs have been performed automatically, so using sorting capabilities of the report viever, we can easily find the optimal runs. Sometimes the differences in evaluation are very small, and the user may wish to examine the non-optimal allocation. For such purposes the simulator allows us to perform a step by step manual run of the designed net. An estimated time and cost of the run completion for every resource-to-requirement assignment at currently chosen state is presented. Thanks to this we know, which consequences are a result of our decision. The simulator remembers all the choices (resource assignments, order of activities start and end times) that a user makes during a simulation. It uses this knowledge to find all the statistical runs that are applicable to this situation, and then provides us with the estimates. Note, that in neither of Tibco, iGraphix nor Yasper, such manual interference in resource assignments is possible, because all resources in these simulators can have only one role. Moreover, up to the authors knowledge, there is no other tool, which supports simulation of resources with multiple roles, giving the user a chance to take part in a simulation at such informative level.

Let us suppose that a simulation has come up to a situation shown on the Fig. 6. The tokens presence on places *Room A* and *Room B* means that the *Initialization* phase has already ended. As we can see, enabled transitions have double-lined border. At the moment both *A:Plaster* and *B:Plaster* transitions require the same skills (in our case: *plasterer*), which have been defined in the corresponding places: *A:[[plasterer]]* and *B:[[plasterer]]*. Both places contain no tokens, which means, that no resources have been assigned to these activities yet. Currently selected place *A:[[plasterer]]* shows in the *Properties View* all the possible resource assignments for the *plasterer* skill. As it turns out, only *Steven* is suitable. Next to his name there is some information indicating the estimated cost and time of the run completion, with him taking part in this activity. In our example, *Steven's Cost* : $[1038 - 42.14]$ 970|1034|1050|1050|1084 *Time* : $[63 - 9.44]$ 50|56|70|70|72 means:

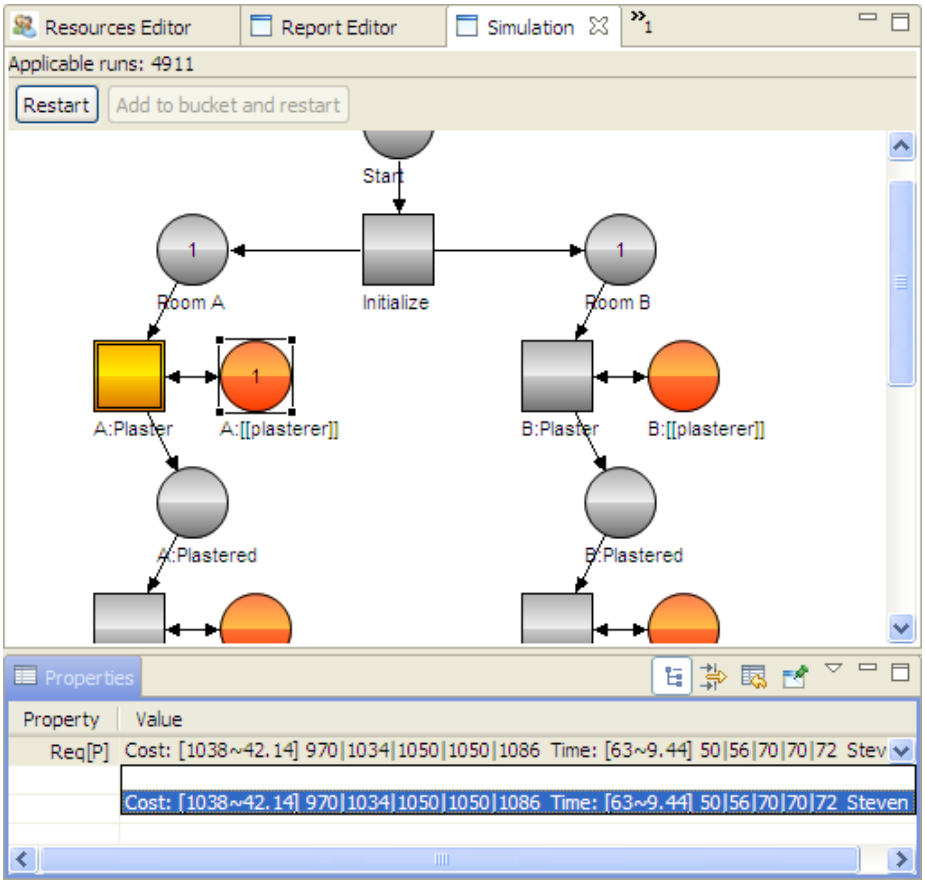


Fig. 6. Simulator

- average cost of completion is 1038 units with standard deviation of 42,14;
- average costs of completion in five successive quantiles are: 970, 1034, 1050, 1050, 1084;
- average time of completion is 63 units with standard deviation of 9.44;
- average times of completion in five successive quantiles are: 50, 56, 70, 70, 72.

When we select the *B:[[plasterer]]* place, the estimates for *Steven's* change to $Cost: [1004 - 44.44]\ 950|988|1000|1011|1075$ $Time: [65 - 8.73]\ 60|60|60|69|80$. So, starting work in *room A* will cost us more, but it will shorten the total execution time. Knowing this at such an early stage lets us undertake proper decisions from the very beginning. On that basis we may tend to choose *room A* first, if we want to save time. On the other hand, when cost is being considered crucial, we should choose beginning work from *room B*.

When all requirements are met, the corresponding activity becomes active. We can fire such enabled transition, and move to the next activity, where we will have more decisions to make. Note, that at every time, we make a decision or fire a transition, the estimates are being recalculated, to show the current situation in the net. This rule also applies to the *Report Viewer*.

Report Viewer Much more information about current situation in the simulated net can be seen on additional report view. The sample screenshot on Fig. 7 presents the state of report viewer, when *Steven* has been assigned to *A:Plaster* activity in the simulator. Thanks to synchronization between these two tabs, we can always take a look at all the computed details and expected values. Diagrams on the 7 present:

- **Total cost distribution** — average, cumulative cost of run performance in current simulator situation in 10 successive quantiles;
- **Resources cost distribution** — as above, but for each of the resource;
- **Total time distribution** — average time of run performance in current simulator situation in 10 successive quantiles;
- **Resources time distribution** — as above, but for each of the resource.

Diagrams of total cost and time provide us with information about differences between optimistic and pessimistic run executions. When viewing costs of resources, we can see, which of them have major influence on the growth in pessimistic cases. In the example shown above it is a fact that, when considering cost, in the optimistic case (on the left side of the diagram), *Steven* involvement is minimal, and the cost of *Steven's* work even goes below the cost of *Tom's* work. On that basis we can conclude that the main way to cut the cost will be to maximize *Tom's* involvement and minimize *Steven's*. It can be seen that bigger involvement of Steven means smaller involvement of Tom and vice versa. So if time does not matter we will prefer the cheaper resource.

The distribution of resource involvement time gives us somewhat different conclusions. It cannot be explicitly determined whether one of the resources should be favoured to improve the process duration. It may seem at a first glance, that in order to optimize the run we should always choose the fastest resource. But it is not the case, since it can slow down other parts of the workflow. The faster one can be the only one who can perform another action which should not be delayed.

Two additional tables at the bottom of this view present:

- list of runs, which are up to date with current situation in the simulator (including time and cost of them as well as pointing out the three most busy resources);
- list of successive concurrent events for the run selected in the first table.

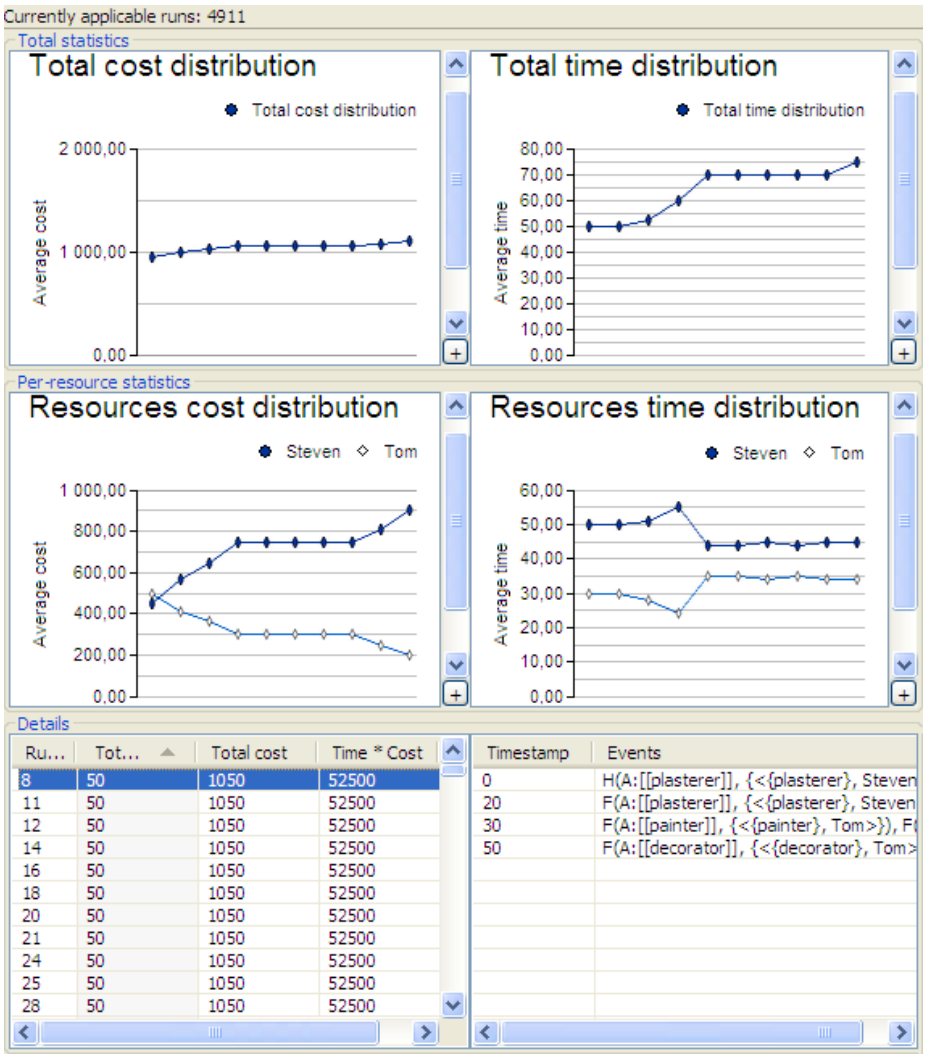


Fig. 7. Report viewer

Bucket Editor To ease the efforts, and allow the user to store the most interesting runs for further analysis, a simple *Bucket Editor* has been created. After the simulation comes to an end, the user can store the run he has just performed manually. This run will be available for later view. The bucket presents information similar to the report viewer, but because it contains information for several, manually chosen runs (not thousands like report viewer), those can be presented in a slightly different form.

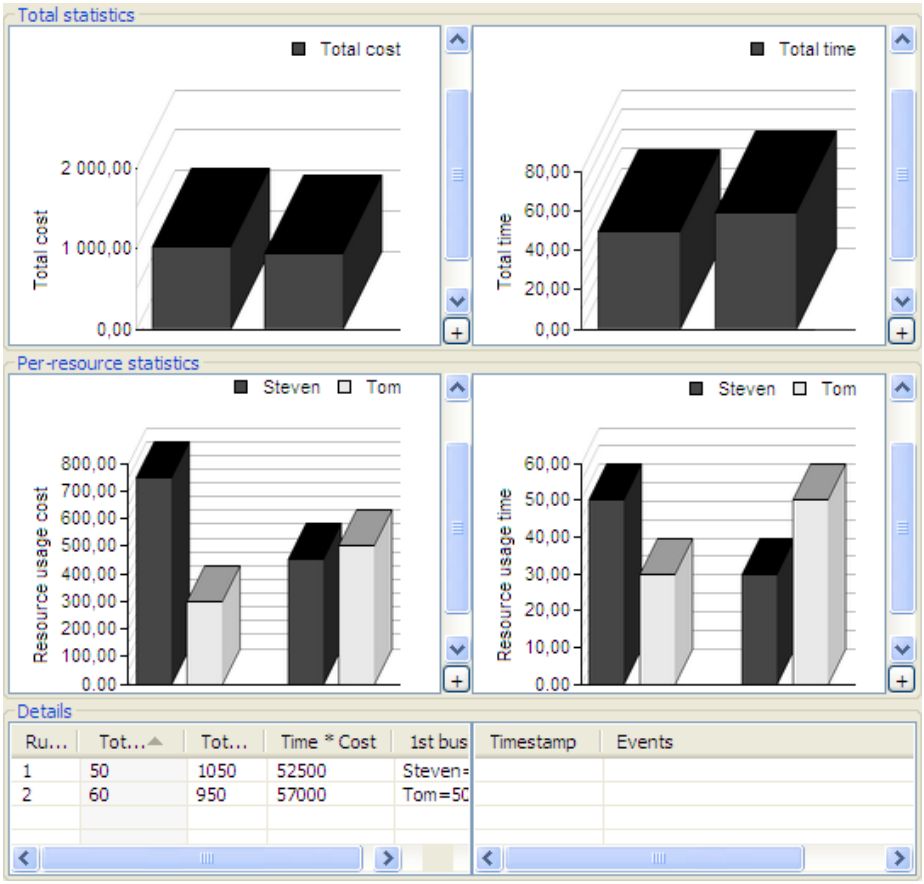


Fig. 8. Bucket viewer

Fig. 8 shows a screenshot of the bucket with two stored runs. As we can see it is organized similarly to the report viever, and the visible diagrams present the same information as those from report viewer:

- **Total cost** — total cost of execution each of the stored runs separately;
- **Resources cost distribution** — as above, but for each of resources;

– **Total time distribution** — total time required to complete each of the stored runs separately;
– **Resources time distribution** — as above, but for each of resources.

The bottom tables represent information about stored runs, showing exactly the same attributes and properties, as the report viewer does.

First of them (displayed on the left hand side of every diagram) has been performed with the goal to minimize the time. Second, though, was made to run the process as cheap as possible, ignoring the time at all. As we can see, the cost reduction requires extra time in workflow execution. Of course, increasing the execution speed increases the cost. Major differences can also be seen in the resource usage distribution in both of these cases. Cost reduction has led to a significant decrease of *Steven's* involvement. In this case, *Tom* takes some of his responsibilities. When the time was the goal, the proportions has significantly changed. *Steven* became more desired, because *Tom* seemed cause the most delays.

4 Conclusions and future work

The provided example shows, how largely the work organization can differ, depending on goals that we want to achieve. In many cases minor changes in the resource allocation can have large influence on the overall result and, conversely, sometimes major allocation changes result in very similar outcomes. Very often humans do not take under consideration all the aspects, which the presented tool does. Its role is to simplify this whole process and make it easier, to find a suitable organization plan that will fit our needs and fulfill the criteria. While the simplification of resource allocation planning is one point, the other one is the possibility to point out uncommon executions, which can lead to increased effectiveness and so increase our profits. Combining user's knowledge and computer's computation power could therefore lead to major design corrections, which would be a basis for further business improvements.

The presented tool is still a prototype, and there are some practical and theoretical issues that need to be addressed. On the theoretical side additional extensions to resource and workflow definition language should be introduced. This includes further conformance to the form of resources described in [BPS09], because only a few resource attributes have already been implemented in this tool. There is also a big potential of the currently used workflow language, which can be further extended. In the context of statistical data tooling, some additional information could be presented (e.g. the Student's t-distribution, $3 - \sigma$). Finally, no measure of reliability have yet been introduced and, in the sense of realism of modelled cases, this is one of the major flaws of the presented tool.

On the more practical side, the integration with existing systems and methods should be made. First of all the statistical runs of the net could generate logs from the run to make them readable by other applications in the same way as the YAWL does [WS09]. The conformance with CPN seems attractive, because of its formal basis and constant development. Secondly, the integration with currently

used common tools is to be made as well. The LDAP for instance is a main source for human resource information. The reporting tools like Microsoft Dynamics AX (Microsoft Business Solutions – Axapta) can be a source of information about resource experience and effectivness of each of such resource. Any other form of gathering of knowledge about past and predictable future resource usefulness can become important in such case. Therefore more integration of this tool will be examinated and developed in the future.

Acknowledgment

The authors would like to thank the reviewers of this work for the effort they put into improving the paper by the constructive reviews, which resulted in a thorough revision of the paper.

Tool description

The tool is written in Java and can be run on any machine with Java (JRE 6.0) installed. It can be downloaded from the http://duch.mimuw.edu.pl/~pch/IRSMT/irsmt.zip. The zip file contains a full version of the tool and a README file, which explains how to install and use the application.

References

[PCh03] Piotr Chrząstowski-Wachtel, Boualem Benatallah, Rachid Hamadi, Milton O'Dell, Adi Susanto, *Top-down Petri Net Based Approach to Dynamic Workflow Modelling*, Lecture Note in Computer Science. v2678. 336-353., 2003.

[BPM05] M. Laugna, J. Marklund. *Business Process Modeling, Simulation, and Design.* Prentice Hall, Upper Saddle River, New Jersey, 2005.

[YA06] Kees van Hee, Olivia Oanea, Reinier Post, Lou Somers, Jan Martijn van der Werf, *Yasper: a tool for workflow modeling and analysis*, Application of Concurrency to System Design, International Conference on, pp. 279-282, Sixth International Conference on Application of Concurrency to System Design (ACSD'06), 2006.

[BS07] Bruce Silver, Bruce Silver Associates, *The BPMS Report: TIBCO iProcess Suite 10.6*, BPMS Watch www.brsilver.com/wordpress, 2007.

[BPS09] W.M.P. van der Aalst, J. Nakatumba, A. Rozinat, and N. Russell. *Business Process Simulation: How to get it right?* In J. vom Brocke and M. Rosemann, editors, International Handbook on Business Process Management, Springer-Verlag, Berlin, 2009.

[WS09] A. Rozinat, M. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge., *Workflow Simulation for Operational Decision Support.*, Data and Knowledge Engineering, 68(9):834-850, 2009.

Detecting and Repairing Unintentional Change in In-use Data in Concurrent Workflow Management System

Phan Thi Thanh Huyen and Koichiro Ochimizu

School of Information Science, Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa, 923-1292, Japan {huyenttp, ochimizu}@jaist.ac.jp

Abstract. Workflow verification has attracted a lot of attention, especially control flow aspect. However, little research has been carried out on data verification in workflow literature although data is one of the most important aspects of workflow. This paper proposes an approach for detecting and repairing **Unintentional Change in In-use Data** (*UCID*) in a Concurrent Workflow Management System at build time. We define UCID as a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers due to non-deterministic access to shared data by different activities. Differently from previous studies, we consider UCID in two different ways: between concurrent activities in a single workflow (*intra-UCID*) and between activities in different concurrent workflows (*inter-UCID*). In this paper, we first investigate UCID situations in a workflow management system, and then we define a Time Data Workflow, an extension of the WF-Nets with time and data factors, with many attributes supporting UCID detection and correction. Based on these definitions, we develop an algorithm which helps to detect potential intra/inter-UCID at build time, along with algorithm evaluation and UCID resolution methods. Finally, we introduce a concrete project on building a change support environment for cooperative software development using UCID theory.

Keywords: Unintentional Change in In-use Data, Time Data Workflow, concurrent workflows, algorithm, Workflow Nets

1 Introduction

Correctness of a workflow model is very important, because any errors in workflow can lead to execution failure of the corresponding process. Therefore, workflow should be verified carefully before execution to reduce risks to the target process. Workflow verification has received a lot of attention since the birth of the workflow concept. However, researchers have only focused on structure verification, temporal verification and resource verification [2] [4] [7] [9]. Most verification techniques ignore data aspect and there is little support for data flow verification. Previous works on the data flow aspect have concentrated on detecting common data flow errors such as missing data, redundant data, inconsistent data, garbage data, etc. Among them, Unintentional Change in In-use Data (UCID) is perhaps one of the most dangerous

and common problems. We define UCID as a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers due to non-deterministic access to shared data by different activities. Assuming that workflow is free of control errors, and activities in workflow can be scheduled within temporal constraint, we aim to support data verification in the workflow model by concentrating on UCID detection and correction.

Existing approaches have addressed this problem by detecting potential UCID patterns, limited to concurrent activities of a single workflow. Unfortunately, this error can cross a single workflow boundary. In a Workflow Management System (WFMS), in fact, there exist many workflows executing at the same time, which we call *Concurrent Workflows*, and they may be correlated if two activities from different workflows use shared data. Even if the data flow of each workflow is correct, we cannot ensure correctness of the whole system because of the mutual interactions among workflows. The problem is how to detect non-deterministic access to shared data of activities belonging to not only the same workflow but also different workflows and how to repair this kind of data abnormality.

Reference [19] is our first efforts in handling the UCID problem. Potential UCID situations, Time Data Workflow (TDW) concepts, along with two algorithms for detecting intra-UCID and inter-UCID have been introduced in [19]. This paper is a refined and extended version of the [19]. In this paper, we redefine TDW as an extension of Workflow Nets (WF-Nets) [8] instead of Petri Nets as before. Based on these definitions and two algorithms for detecting intra/inter-UCID in [19], a revised version of UCID detection algorithm is built. Compared with the previous ones, this revised algorithm is more accurate and useful. Furthermore, some heuristics for making the algorithm more flexible and effective are discussed. UCID resolution methods are also proposed in this paper. Then, we illustrate this theory in practice by using it in designing workflows which represent change activities in a software change process.

Our approach in UCID detection is to observe behaviors of concurrent activities having data relation. In the case of activities in the same workflow, their total orders can be decided based on control flow. However, control flow does not help in the case of activities in different workflows. Therefore, we must use activities' execution time attribute to identify their total orders. Regarding UCID resolution, we take advantage of composition features of the Petri Nets to create new workflows with UCIDs removed.

The rest of this paper is organized as follows. Section 2 discusses the motivation of our research. Section 3 defines the Time Data Workflow (TDW), an extension of the Workflow Net with time and data factors. Section 4 introduces UCID situations caused by concurrent activities in the same workflow (*intra-UCID*) or activities in different concurrent workflows (*inter-UCID*) [19]. An algorithm for detecting potential UCID in both cases of intra/inter-UCID at build time, along with algorithm evaluation, is given in Section 5. Section 6 presents UCID resolution methods. Section 7 introduces our project on building a change support environment for cooperative software development. Theory about UCID problem is employed in this project to detect and repair data abnormalities among concurrent Change Support

Workflows. Section 8 reports on related work and finally, Section 9 concludes the paper and discusses points to future work.

2 Motivation

Let's take an example. We have two workflows W_1 and W_2, which are being executed independently. Workflow activities are modeled by rectangles, and data modified by an activity are written inside the corresponding rectangle. A small arrow is attached to a rectangle to denote an activity which is being executed. Data of the system are stored in a central repository. W_1 has five activities which modify A, X, B, C and D respectively. B and D are modified based on the value of X created by A_{12}. W_2 also has five activities which modify E, X, F, G and H respectively. Both A_{12} and A_{22} will modify X, but designers of W_1 and W_2, who don't have a comprehensive view of the whole system, may not recognize this problem. This is a common problem, especially in a big system with many workflows.

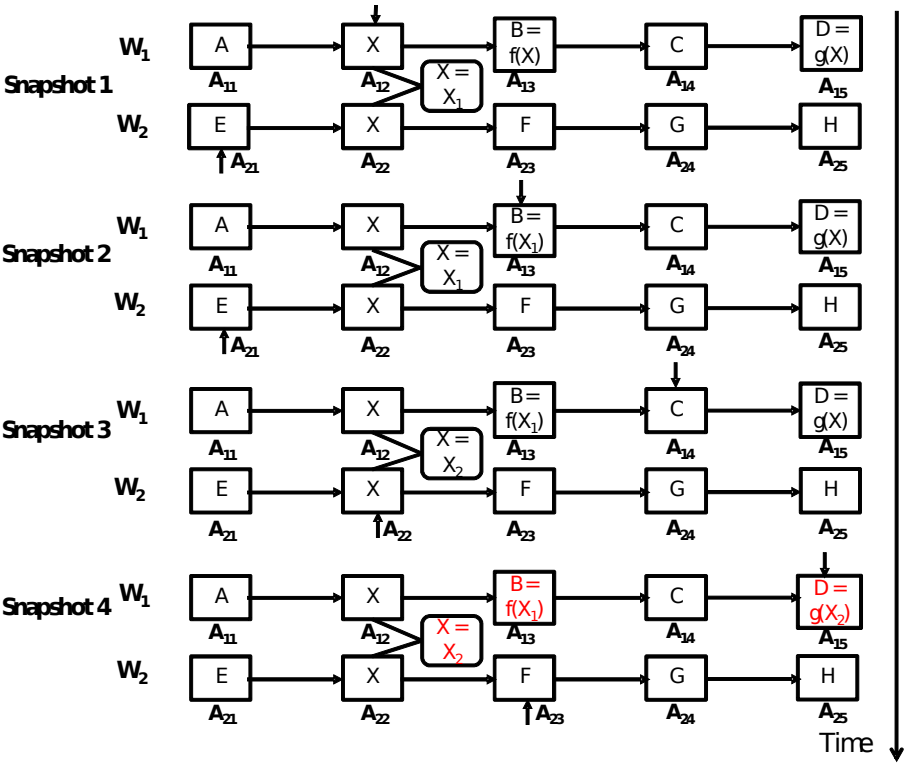


Fig. 1. Motivating example

Figure 1 describes some snapshots of the system at different time. For simplicity, we concentrate on describing the change in value of data elements relating to shared

data X. In snapshot 1, A_{12} changes value of X to X_1. In snapshot 2, A_{13} changes value of B based on the value of X, X_1. In the next snapshot, A_{22} changes value of X from X_1 to X_2. In the last snapshot, A_{15} changes value of D based on the current value of X which is X_2. If X_1 is different from X_2, there are two problems in this scenario: X_1 is lost and D is assigned an unexpected value because D is modified based on the value X_2 instead of the value created by activity A_{12}, X_1. This is different from the intentions of the designers of the workflow W_1 and may cause an inconsistency between B and D. Regarding our definition of UCID, these errors are categorized into inter-UCID errors.

The first problem is similar to the lost update problem in database theory. Lost update problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect [20]. In this case, version control systems (VCSs) can be used if data of the system are individual artifacts like documents, source codes, etc. Version control is the management of changes to documents, programs, and other information stored as computer files. Changes are usually identified by a number or letter code, termed the "revision number". Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

Unfortunately, VCS cannot help to avoid the second problem. In this situation, if data of the system are stored in a central database, the database management system (DBMS) can provide some concurrency control techniques, which are used to ensure the noninterference or isolation property of concurrently executing transactions such as locking techniques, timestamp ordering based techniques, etc. A database transaction is a transaction which satisfies the ACID (atomicity, consistency, isolation and durability) properties. These properties should be enforced by the concurrency control and recovery methods of the DBMS [20]. However, in this method, we must specify the boundary of each transaction. This requirement is difficult to implement because there are many people involved in a workflow and people in a workflow may know nothing about other workflows. If the whole workflow is considered as a unique database transaction, it is impractical because a workflow may use many data elements and may happen for a long time.

If this type of errors is discovered at runtime, a recovery mechanism must be performed to ensure the correctness of the whole system. However, recovery is a rather expensive work, especially in a cooperative environment with many concurrently executing workflows. Therefore, detecting these errors as soon as possible is necessary to reduce risk to the target process.

This paper examines UCID situations in a general basic system without concerning which type of workflow data is stored in the central repository of the system and the implementation of the central repository as well.

Regarding inter-UCID, our problem domain is workflows whose data and estimated execution time can be decided at the design phase, for example workflows in the software evolution process. In these cases, an early UCID detection will help workflow designers to have a more comprehensive view of the system, and make timely adjustments to the original workflows to avoid error at runtime. We assume

that the following steps are conducted before workflow execution: identifying workflow activities and their orders, assigning activity properties (data, time…), and checking error using UCID detection and correction theory. If some potential UCID errors are detected, the first and second steps should be re-executed, based on suggested solutions given by UCID detection system.

With reference to workflows in which estimated execution time is not available at design time, UCID patterns and detection method will be used to detect UCID errors from workflow execution histories. However, this is out of the scope of this paper.

3 Time Data Workflow (TDW)

There are many ways to model a workflow, such as directed graphs, UML activity diagram, PERT, etc. In this paper, we chose the WF-Nets based approach to model workflow process, because it has many useful features needed in the area of business process modeling besides the mathematical nature of the underlying Petri Nets formalism [17].

WF-Nets is a subclass of Petri Nets dedicated for process/workflow modeling and analysis. Petri Nets is a popular graphical and mathematical modeling language in describing and analyzing systems which are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic [17]. Formally, Petri Nets is a tuple PN = (P, T, F) where P is a finite set of places, T is a finite set of transitions ($P \cap T = \emptyset$) and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation) [8]. A Petri Nets PN = <P, T, F> is a WF-Nets if and only if there is one source place $i \in P$, one sink place $o \in P$ such that $\bullet i = \emptyset$, $o\bullet = \emptyset$, and every node $x \in P \cup T$ is on a path from i to o [8].

Our Time Data Workflow (TDW) is an extension of WF-Nets with time and data factors. Time and data are represented as attributes of transitions in a TDW. In this paper, we consider two types of relationships between an activity and a data element. First, an activity may *read* a particular data element as its input data. Second, an activity may *write* a particular data element as its output data. This means that this data element is assigned a new value. Inside an activity, *read* always happens before *write*. Assuming that durations of activities can be estimated at build time, we augment each activity A with two time values *min(A), max(A)* which describe the minimum and maximum execution durations of A respectively. The time unit is selected depending on specific workflow applications. Based on reference point P, which is the start time of its corresponding workflow, we can infer the *Earliest Start Time, EST(A)*, and the *Latest Finish Time*, *LFT(A)*, of A at run time. If *S(A)*, *F(A)* are the *Start Time* and *Finish Time* of this activity at run time respectively, we can conclude that the *Active Interval* of A, *[S(A), F(A)]*, is within its *Estimated Active Interval, [EST(A), LFT(A)]*, that is, $[S(A), F(A)] \subseteq [EST(A), LFT(A)]$ [19].

In a TDW, activities are modeled by transitions, and causal dependencies are modeled by places and arcs, as shown in Figure 2 [19]. Building blocks such as the AND-split, AND-join, OR-split, OR-join are used to model sequential, conditional, parallel and iterative control structures of workflows. AND-split and OR-split transition correspond to transitions with two or more output places, while AND-join

and OR-join transition correspond to transitions with multiple incoming arcs. Different symbols are attached to original rectangles to distinguish normal transitions from transitions containing branching conditions. Figure 2a illustrates a typical transition in a TDW, with execution duration ranging from d1 to d2; data elements a, b are inputs and c, d, e are outputs. The other parts of Figure 2 show how basic constructions of a workflow are represented by TDW's notations [19]. For the sake of simplicity, each activity is represented by a transition. Therefore, the terms 'activity' and 'transition' are interchangeably used in this paper.

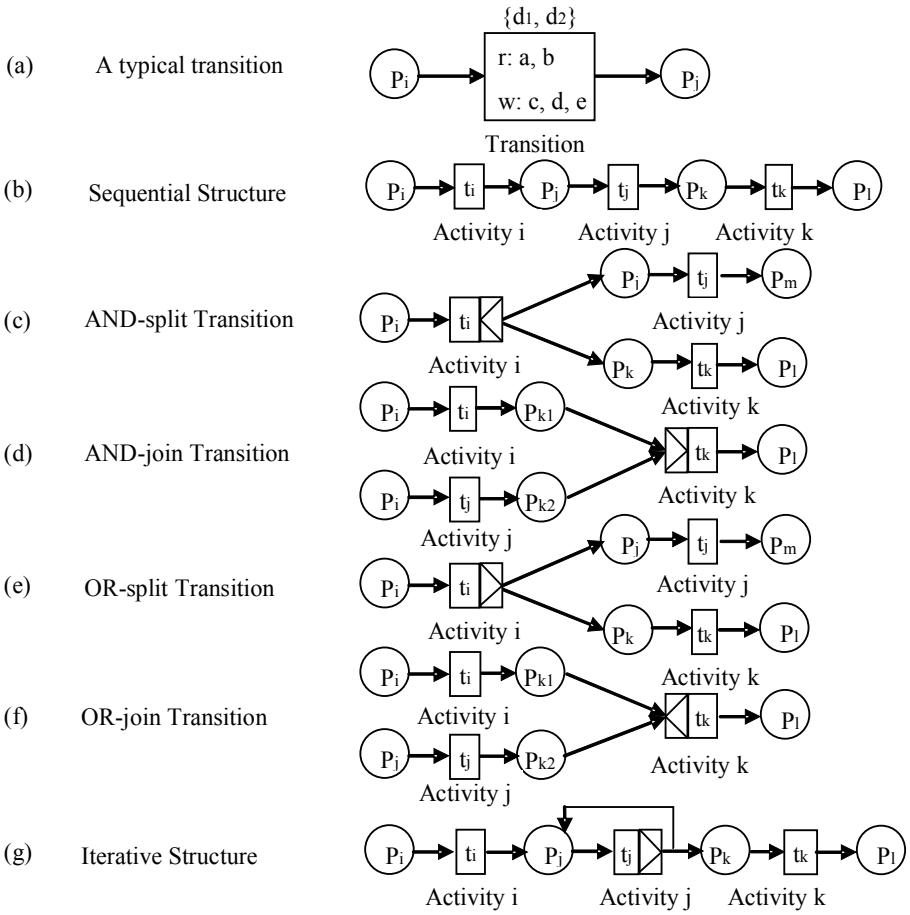


Fig. 2. Workflow primitives specified by TDW

As an extension of WF-Nets, TDW specifies the time and data properties of a single case in isolation, assuming that different cases are completely independent from each other. Therefore, UCIDs are caused by activities in a single TDW instance or activities belonging to workflow instances of different TDWs. Without the loss of generality, we assume that each TDW has one instance only.

Definition 1 (Time Data Workflow – TDW) A TDW, w, is a tuple <P, T, F, id, D, R, DE, TI > where:

<P, T, F> is a WF-Nets with places P, transitions T and arcs F

id is the workflow identifier.

D is a set of data elements.

R = {r, w, u} is a set of possible access rights to data elements (r: ***read***, w: ***write***, u: ***use*** (either ***read*** or ***write***)).

DE: T x R→ 2^D is a function that returns a set of data elements associated with a transition and an access right.

TI: T → R^+ x (R^+ x ∞) is a time interval function that returns minimum and maximum execution durations of a transition.

Definition 2 (Concurrent Time Data Workflow Model) A Concurrent TDW Model cwm = (W, T_{wm}) is a collection of TDWs which have overlapping execution times (concurrent TDWs):

W = {w_1, w_2… w_n} is a set of concurrent TDWs, where w_i = < P, T, F, id, D, R, DE, TI >.

T_{cwm} = T(w_1) ∪T(w_2) ∪ …∪T(w_n) is the set of all transitions (activities) in cwm.

Given a TDW w as in Definition 1, we have the following definitions [19]:

Definition 3 (Path) A Path is a sequence of consecutive arcs.
A sequence p = (x_o, x_1, …, x_k) is a Path iff ∀i, 0 < i < k – 1: (x_i, x_{i+1}) ∈F

Definition 4 (Transition Path) A sequence p = (t_0, p_1, t_1,… , t_k) is a Transition Path iff it is a path and t_0, t_k ∈ T.

Definition 5 (Transition Reachability) Transition t_i is reachable from t_j if there exists a transition path (t_i,... , t_j) on wm.
Reachable (t_i, t_j) = true iff ∃transition path p = (t_i,... , t_j)

Definition 6 (Transition Distance) Given two transitions t_i, t_j where Reachable (t_i, t_j) = true or where Reachable (t_j, t_i) = true, the Transition Distance between t_i, t_j is the length of the shortest path between them.

Definition 7 (Nearest Common Transition) Given two transitions t_i, t_j where Reachable (t_i, t_j) = false and where Reachable (t_j, t_i) = false, their Nearest Common Transition is the common transition which has the shortest distances to both of them, denoted as t_{nct}.

Definition 8 (Closest Data Relation Transition) Given two transitions t_i, t_j, where their nearest common transition is not an OR-split transition, t_j is called the Closest Data Relation Transition of t_i on data element d if t_j just precedes t_i in terms of time, and both t_j and t_i *use* (*read*/*write*) d, denoted as t_{cdrt}.

4 UCIDs in a Concurrent TDW Management System

A Concurrent TDW Management System is a workflow management system which is responsible for TDW construction and management. A module of UCID detection

and correction is also integrated into this system.
Data flow can be implemented explicitly as a part of the workflow model by using a separate channel to pass data from one activity to another. Otherwise, it can also be implemented implicitly through a control flow or process data store [3]. The process data store is basically a central repository where all workflows' activities can access or update their data. We choose implicit data flow through the process data store as a basis for our approach. In this implementation model, UCID may occur, particularly in cases involving concurrent execution paths.

Given a Concurrent TDW Model cwm as in Definition 2, we have the following definitions:

Definition 9 (Data Relation) Two activities a_i, a_j ($i \neq j$) have data relation if $DE(a_i, u) \cap DE(a_j, u) \neq \emptyset$ [19].

Definition 10 (Concurrent activities) Two activities are called concurrent activities iff they belong to two parallel branches of a TDW or they are in different TDWs and have overlapping Active Intervals.

Definition 11 (Unintentional Change in In-use Data) A situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers due to non-deterministic access to shared data by different activities [19].

Here we distinguish two kinds of UCID: intra-UCID and inter-UCID. The former considers UCID situations concerning concurrent activities in the same workflow, while the latter is related to concurrent activities in different workflows. Definition 12, 13 are based on definitions of read-write conflict and write-write conflict in [1].

Definition 12 (RW Intra-UCID) A situation in which an activity A tries to ***read*** data from a shared variable x and an activity B tries to ***write*** data to the same shared variable *x* and vice versa, where A, B are concurrent activities in the same workflow.

Definition 13 (WW Intra-UCID) A situation in which two concurrent activities in the same workflow, A and B, try to ***write*** data to the same shared variable.

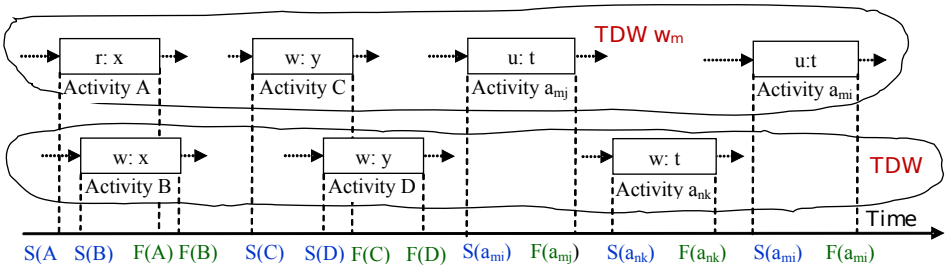


Fig. 3. Inter-UCIDs

Definition 14 (RW Inter-UCID) A situation in which an activity A tries to ***read*** data from a shared variable x and an activity B tries to ***write*** data to the same shared variable *x* and vice versa, where A, B are in different concurrent workflows and have overlapping Active Interval ($[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$).

Definition 15 (WW Inter-UCID) A situation in which two activities A and B try to ***write*** data to the same shared variable, where A, B are in different concurrent

workflows and have overlapping Active Interval ($[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$).

Definition 16 (UWU Inter-UCID) A situation in which there are inconsistent views of shared data by two activities in the same workflow, because their shared data are ***written*** externally by an activity in a different concurrent workflow.

As depicted in Figure 3, two activities a_{mi}, a_{mj} of TDW w_m ***use*** (***read*** or ***write***) data element t, where a_{mj} is the closest to a_{mi} in terms of time and $F(a_{mj}) < S(a_{mi})$, which means $t_{cdrt}(a_{mi}, t) = a_{mj}$. A UWU Inter-UCID happens because activity a_{nk} of a different workflow w_n ***writes*** to t within the time interval $[F(a_{mj}), S(a_{mi})]$. RW Inter-UCID and WW Inter-UCID also happen between activity A and activity B, activity C and activity D respectively.

5 Detection of Potential UCID in a Concurrent TDW Management System

Regarding UCID definitions, inter-UCIDs are identified based on the Active Interval of activities having data relation. However, Active Interval of an activity can only be determined at runtime when it has finished its execution, and hence Estimated Active Interval is used instead of Active Interval to find potential UCID at build time, before a new TDW is put into the Concurrent TDW Management System to start.

5.1 Calculation of Estimated Active Interval [19]

Designating the start time of a TDW w as a reference point, P_w, we can infer the Estimated Active Interval of an activity A [EST(A), LFT(A)] with respect to its minimum and maximum executing durations {min(A), max(A)} and basic control structures.

Let us say that A_s is the Start activity of a TDW w, then we have $EST(A_s) = P_w$ and $LFT(A_s) = P_w + max(A_s)$. For executing activity A, EST(A) = S(A) and LFT(A) = F(A) if A has been completed.

Sequential Connection (Figure 2b)

$EST(A_j) = EST(A_i) + min(A_i)$; $LFT(A_j) = LFT(A_i) + max(A_j)$

AND-Split Connection (Figure 2c)

$EST(A_j) = EST(A_i) + min(A_i)$; $LFT(A_j) = LFT(A_i) + max(A_j)$

$EST(A_k) = EST(A_i) + min(A_i)$; $LFT(A_k) = LFT(A_i) + max(A_k)$

AND-joint Connection (Figure 2d)

$EST(A_k) = MAX\{EST(A_i) + min(A_i); EST(A_j) + min(A_j)\}$

$LFT(A_k) = MAX\{ LFT(A_i), LFT(A_j)\} + max(A_k)$

OR-Split Connection (Figure 2e)

$EST(A_j) = EST(A_i) + min(A_i)$; $LFT(A_j) = LFT(A_i) + max(A_j)$

$EST(A_k) = EST(A_i) + min(A_i)$; $LFT(A_k) = LFT(A_i) + max(A_k)$

OR-joint Connection (Figure 2f)

$EST(A_k) = MIN\{EST(A_i) + min(A_i); EST(A_j) + min(A_j)\}$

$LFT(A_k) = MAX\{ LFT(A_i), LFT(A_j)\} + max(A_k)$

5.2 Potential UCID Detection Algorithm

Given a Concurrent TDW Model cwm = (W, T_{cwm}), where W = {w_1, w_2, …, w_k} and T_{cwm} = T(w_1) ∪T(w_2) ∪ …∪T(w_k), w = <P, T, F, id, D, R, DE, TI>. The main idea of this algorithm is to select one activity and compare it with the other activities. If two activities have data relation, we will check if there is a potential UCID. In the case of concurrent activities in the same workflow, potential intra-UCIDs can be detected with respect to Definitions 12, 13. If two compared activities are in different workflows and have overlapping Estimated Time Intervals, there is a possibility of an RW/WW inter-UCID occurrence (Definitions 14, 15). If only the data relation exists and one activity occurs before the other, we will compare this situation with the definition 16 and the pattern in Figure 3 to find out a potential UWU inter-UCID.

Step 1: Initialization:

1.1 Let *S* be a set of unchecked activities. S is initialized with all unfinished activities of T_{cwm};

1.2 Calculate Estimated Active Interval for all activities in S;

1.3 flag = TRUE is a Boolean variable*;*

Step 2: For every pairwise of activities (a_{mi}, a_{nk}) in S, execute the following steps:

2.1 Check their Data Relation

Let U_{mnik} be the set of shared data between a_{mi} and a_{nk}: U_{mnik} = DE*(*a_{mi}, u*)* ∩ DE*(*a_{nk},u);

2.1.1 If U_{mnik} = ∅, a_{mi} and a_{nk} do not have data relation. Therefore UCID cannot happen between a_{mi} and a_{nk};

2.1.2 If U_{mnik} ≠ ∅, a_{mi} and a_{nk} have data relation. Take the next step.

2.2 If a_{mi} and a_{nk} in the same workflow, check intra-UCID possibility. Otherwise, check inter-UCID possibility;

2.2 Check intra-UCID possibility

2.2.1 If a_{mi} and a_{nk} belong to two parallel branches of a workflow, this means that their Nearest Common Transition, denoted as t_{nct} (a_{mi}, a_{nk}), is an AND-split transition, they are concurrent activities. Take the next step;

2.2.2 For every data element, denoted as d_{mnikl}, in U_{mnik}, check the access right to d_{mnikl} of a_{mi} and a_{nk}:

2.2.2.1 If both of them have *write* access right to d_{mnikl}, this means that d_{mnikl}∈DE*(*a_{mi}, w*)* and d_{mnikl}∈DE*(*a_{nk}, w*)*, then flag = FALSE. There is a potential WW Intra-UCID between a_{mi}, a_{nk} on d_{mnikl};

2.2.2.2 If one activity has *write* access right to d_{mnikl} and the other has *read* access right to d_{mnikl}, this means that (d_{mnikl}∈DE*(*a_{mi}, w*)* and d_{mnikl}∈DE*(*a_{nk}, r)) or (d_{mnikl}∈DE*(*a_{mi}, r) and d_{mnikl}∈DE*(*a_{nk}, w)), then flag = FALSE. There is a potential RW Intra-UCID between a_{mi}, a_{nk} on d_{mnikl};

2.3 Check inter-UCID possibility /* Figure 3***/**

2.3.1 If a_{mi} and a_{nk} have overlapping Estimated Active Interval, this means that [EST(a_{mi}), LFT(a_{mi})] ∩ [EST(a_{nk}), LFT(a_{nk})] ≠ ∅, they are potential concurrent activities: check RW/WW inter-UCID possibility. Otherwise, check UWU inter-UCID possibility;

2.3.2 Check potential RW/WW inter-UCID

For every data element, denoted as d_{mnikl}, in U_{mnik}, check the access right to d_{mnikl} of

a_{mi} and a_{nk}:

2.3.2.1 If both of them have *write* access right to d_{mnikl}, this means that $d_{mnikl} \in DE(a_{mi}, w)$ and $d_{mnikl} \in DE(a_{nk}, w)$, then flag = FALSE. There is a potential WW Inter-UCID between a_{mi}, a_{nk} on d_{mnikl};

2.3.2.2 If one activity has *write* access right to d_{mnikl} and the other has *read* access right to d_{mnikl}, this means that $(d_{mnikl} \in DE(a_{mi}, w)$ and $d_{mnikl} \in DE(a_{nk}, r))$ or $(d_{mnikl} \in DE(a_{mi}, r)$ and $d_{mnikl} \in DE(a_{nk}, w))$, then flag = FALSE. There is a potential RW Inter-UCID between a_{mi}, a_{nk} on d_{mnikl};

2.3.3 Check potential UWU inter-UCID

Assume that $LFT(a_{nk}) < EST(a_{mi})$. For each data element, denoted as d_{mnikl}, in U_{mnik} where a_{nk} has *write* access right to d_{mnikl}: $d_{mnikl} \in DE(a_{nk}, w)$, perform the following steps:

2.3.3.1 Find out the Closest Data Relation Transition of a_{mi} on d_{mnikl}, denoted as a_{mj}: $a_{mj} = t_{cdrt}(a_{mi}, d_{mnikl})$. If $a_{mj} = \emptyset$, UWU inter-UCID may not happen;

2.3.3.2 If $[EST(a_{nk}), LFT(a_{nk})] \subset [LFT(a_{mj}), EST(a_{mi})]$, then flag = FALSE. There is a potential UWU Inter-UCID among a_{mi}, a_{mj}, a_{nk} on d_{mnikl};

Step 3: Return flag.

5.3 Algorithm Evaluation

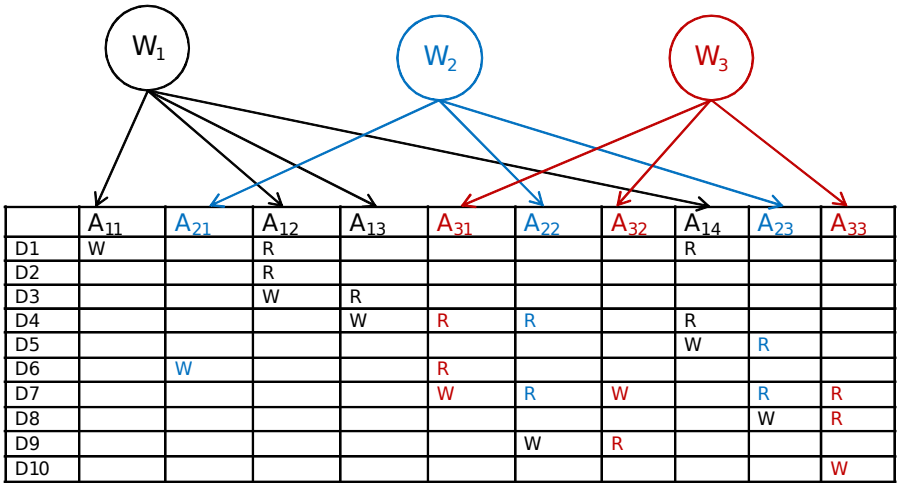
Let's say *n* is the number of unfinished activities in a Concurrent TDW Model *cwm*. In general, we must inspect n^2 combinations of any two unfinished activities to find out some potential UCIDs. This approach allows us to detect not only potential UCID at build time of pre-executed TDWs, but also potential UCID at run time of running TDWs by recalculating the Estimated Active Intervals of their unfinished activities more accurately based on the Active Interval of finished activities. However, depending on applications, we can reduce the number of checking steps by considering some of the following heuristics:

A two dimensional table can be used to record the access right on data elements of activities in a Concurrent TDW Model *cwm*. Figure 4 describes an example of data flow matrix of a Concurrent TDW Model with three TDWs W_1, W_2 and W_3. $\{D_1,\ldots, D_{10}\}$ is the data set of the Concurrent TDW Model. Parallelization can be applied here to reduce execution time. For each element in the data set of *cwm*, there is a thread being responsible for checking potential UCID caused by activities *using* this data element.

After designing a new TDW, UCID check is conducted to find potential UCIDs before this TDW is put into the Concurrent TDW Management System for execution. Let's say *m*, *k*, *l* are the number of unfinished activities in the being considered pre-executed TDW, other pre-executed TDWs, running TDWs respectively, we have $n = m + k + l$. Because the other pre-executed TDWs have been checked in previous examinations, we can skip combinations of two activities in these TDWs to reduce the number of inspected combination to $n^2 - k^2$. If we just want to detect UCIDs caused by activities in the being considered TDW, we will verify m x n activity combinations only. A parallel solution in this case is to create m threads. Each thread will be

responsible for one activity in this TDW and will verify potential UCIDs on combinations created by this activity with the others in different TDWs.

Because potential UCIDs just occur in activities that have shared data, we will verify activities having shared data only. Each data element will store identifications of unfinished activities using it. Therefore, the set of checked activities can be limited to unfinished activities having data relation in the Concurrent TDW Model. If the number of data elements is small, we can start from data elements of the being considered pre-executed TDW to pick out unfinished activities in the Concurrent TDW Model having data relations and use UCID patterns to find out potential errors.



	A_{11}	A_{21}	A_{12}	A_{13}	A_{31}	A_{22}	A_{32}	A_{14}	A_{23}	A_{33}
D1	W		R					R		
D2			R							
D3			W	R						
D4				W	R	R		R		
D5								W	R	
D6		W			R					
D7					W	R	W		R	R
D8									W	R
D9						W	R			
D10										W

Fig. 4. Data flow matrix example

6 Potential UCID Resolution

In general, if potential UCIDs happen, there may be some abnormalities in data flows or control flows of the concerned workflows. A review on the workflow design should be conducted to make sure that this situation is not made on purpose.

Our given solutions in which some of them will change the workflow structure are simply reference models. The final decision will depend on workflow designers to perform modifications that actually lead to a resolved model.

6.1 Potential Intra-UCID Resolution

Potential Intra-UCID may be caused by a mistake of workflow designers in designing parallel branches of a workflow. Therefore, our solution for Intra-UCID is to change the workflow structure by sequentializing or combining error-related activities. Two activities causing potential WW Intra-UCID are merged into one by place/transition fusion (Figure 5a). For RW Intra-UCID, sequentialization is applied to the related

activities. One option is that read activity happens before write activity and the other is that write activity happens before read activity (Figure 5b). Resolution order will begin from WW Intra-UCID cases to RW Intra-UCID cases. With regard to potential UCIDs belonging to the same group, the priority is the happening order.

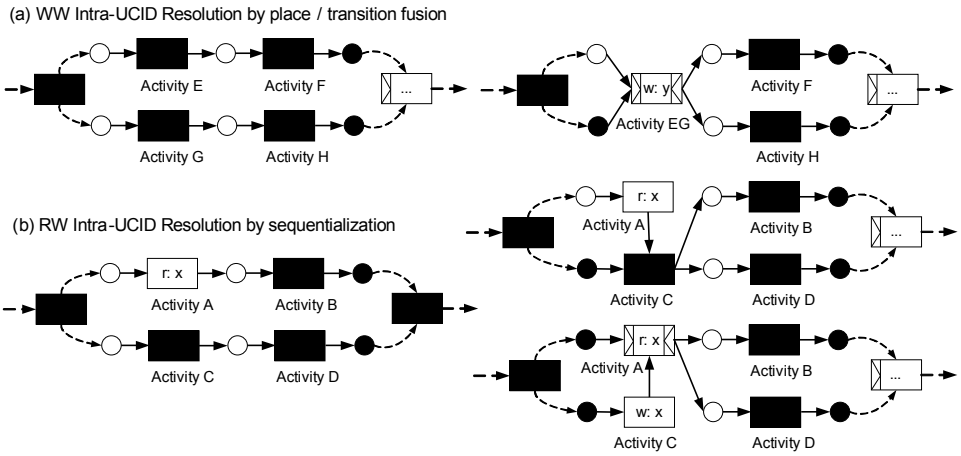


Fig. 5**.** Potential Intra-UCID resolution

6.2 Potential Inter-UCID Resolution

Resolving potential inter-UCID is more complex because workflows are designed for different purposes by different designers and a designer may know nothing about the work of the others. To resolve inter-UCID, the cooperation of different designers is necessary and the result will highly depend on the willingness of designers to communicate with each other.

A method which does not affect the workflow structures is to adjust the workflow schedule by modifying the workflow start time, maximum and minimum execution durations of activities in workflows so that inter-UCID patterns do not occur. Another solution is to change the workflow structure.

First, we will combine related TDWs into one workflow. In order to preserve the structure of the original TDWs, in the new TDW, the Start place connects to an AND-Split transition and the End place is connected to an AND-join transition. Each merged TDW corresponds to a subnet starting from the AND-split transition and ending at the AND-join transition. Because the merged TDWs are started at different times, we insert a Time Start transition between the Start place of each merged TDW and the AND-split transition, a Time End transition between the End place of each merged TDW and the END-join transition. Time activities are just null activities with some duration and they help to merge TDWs without modifying the workflow's schedule seriously. The AND-split transitions, AND-join transitions, Time Start transitions, Time End transition, places and arcs connecting the related workflows together represent the dependency relationships between different workflows which play an important role in the recovery process in the case of workflow failure. They

will not be used to identify the total order of activities in detecting potential intra-UCID in the synthesis TDW. In the case of a running TDW, we can create a new TDW from the original workflow by removing its finished activities, and this new TDW will be combined with other TDWs in a normal way. Another simpler way is to combine the pre-executed TDWs only. After that, workflow designers can adjust the Estimated Active Interval of activities in the new TDW by modifying workflow start time, maximum and minimum execution duration of its activities so that UCID related activities happen after related activities of the running TDW.

Next, we will deal with activities causing potential Inter-UCID. The mechanism to handle potential WW/RW Intra-UCID is applied to WW/RW Inter-UCID cases (Figure 6a, 6b). Regarding UWU potential UCID, three activities related to this error are connected as shown in Figure 6c. If there are many potential Inter-UCIDs between the same two TDWs, the priority is Inter-UCID types (WW > RW > UWU) and occurring time of activities respectively.

As mentioned earlier, inter-UCID resolution is very complex, especially UWU inter-UCID. Currently, our proposed solution is just a reference model which helps workflow managers to have a more comprehensive view of data related workflows. We will try to improve them in the future work.

7 Application

In this section, we present a project on building a change support environment for cooperative software development. UCID theory is used in this project to detect potential UCID between concurrent workflows.

Software systems must be changed under various circumstances during development and after delivery, such as for new requirement, error correction, performance improvement, etc. However, software change is not an easy task, especially in a cooperative environment where software artifacts with very complex dependency relationships are created based on the cooperation of many people. Besides, other problems such as concurrency of works, synchronization of changes on shared artifacts, etc. also make this task more difficult. Therefore, a change support environment is strongly demanded.

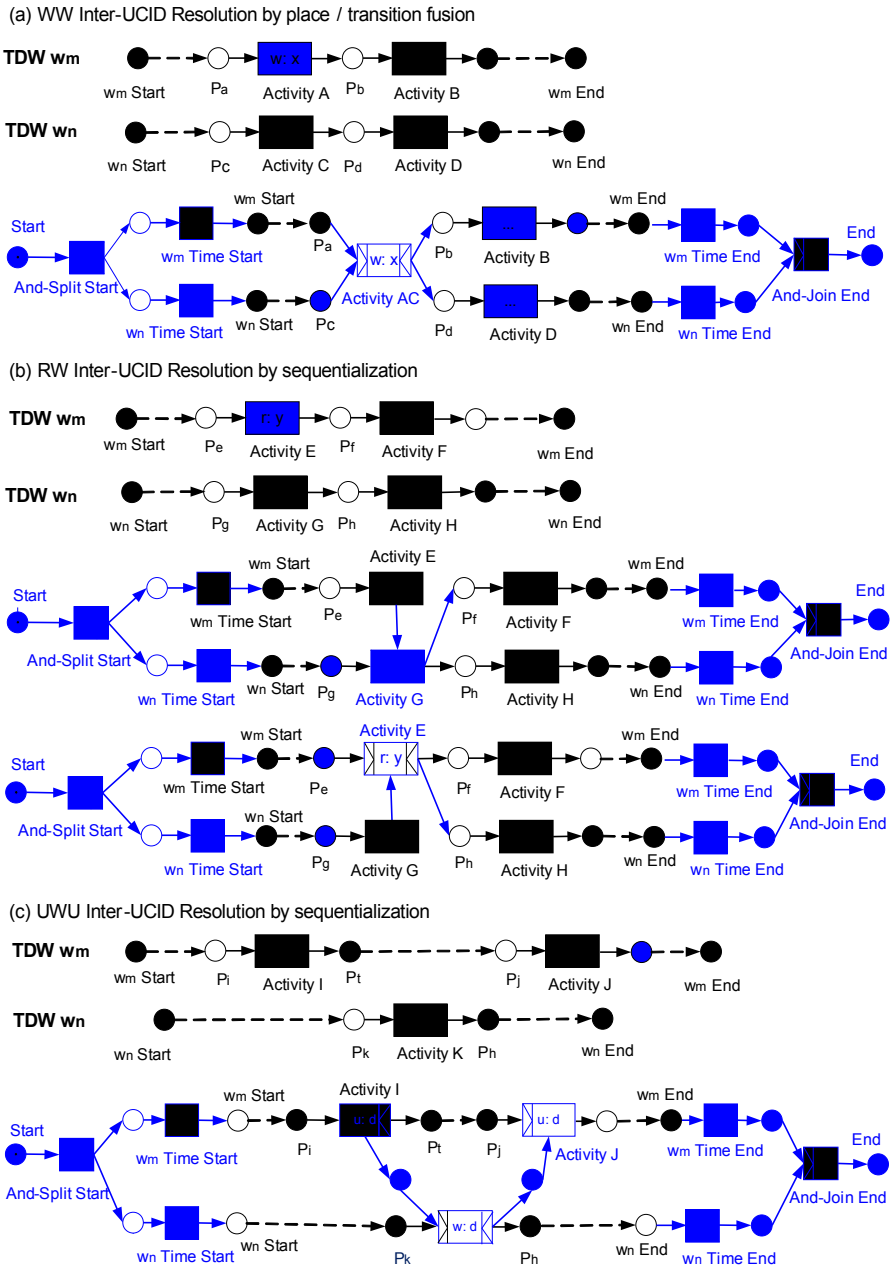


Fig. 6. Potential Inter-UCID resolution

In order to help change workers to perform change activities safety and efficiently in a cooperative environment, we use workflow to represent activities needed to implement a change request. We define Change Support Workflow (CSW) as a sequence of activities required to implement a change. Activities in CSW are

responsible for creating new software artifacts or modifying exiting ones. This means that data elements of CSW are software artifacts which need to be read, modified or created in the change implementation process.

In the first phase of the project, a method for automatically generating dependency relationships among UML elements was given [22]. Change impact analysis which identifies potential consequences of a change can be realized by tracing the generated dependency relationships. Result of this process will be used to generate CSW.

In large and cooperative system, there may be hundreds of CSWs executed at the same time to react to change requirements quickly. However, when there are many CSWs running on the same system, that UML artifacts are shared by different CSWs is unavoidable. If CSWs having shared artifacts are executed at the same time, inconsistencies among their data (UML artifacts) can happen. A version control system is used in our change support environment to deal with data loss; however this system does not help in this situation. Therefore, UCID theory is employed in this project to deal with this problem. Potential UCID can be detected automatically at build time to help workflow designers make timely adjustments to original workflows.

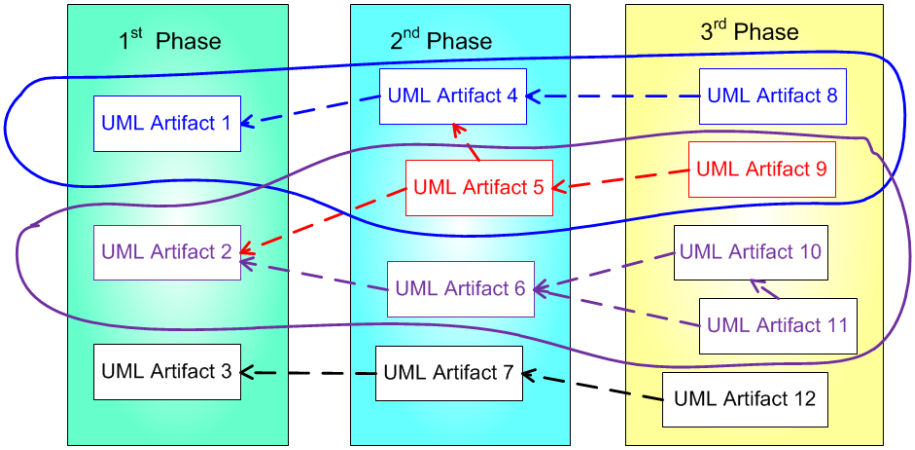


Fig. 7. Example of Relationships between UML Artifacts created during a software development process

Our project supports constructing CSW based on the relationships between impacted UML model elements which are extracted from the result of impact analysis. CSW is modeled by TDW as follows. Each transition corresponds to an activity which creates or modifies at least one UML artifact. Total order of two transitions is identified by examining the dependency relationships between the artifacts modified by these transactions. Access role *write* is assigned to the artifacts which need to be modified or created; the artifacts for reference only are labeled with *read* access role. This draft of CSW will help workflow designers in developing the schedule of the change process. The other steps in developing change schedule such

as estimating activity resources and activity durations will be performed by workflow designers. From Activity Duration Estimates in the schedule, minimum and maximum execution durations of transitions in this CSW can be inferred. To reduce risks at runtime, UCID check on this CSW will be conducted. If some potential UCIDs are reported, data and control structure of this CSW should be adjusted in responding to suggested solutions of the change support system.

Because CSW is constructed based on relationships between software artifacts, potential Intra-UCIDs seldom happen. Besides, if potential UCIDs are reported, the possibility of control flow errors is low too. In this case, workflow designers should review data flow and pay attention to shared data elements among concurrent CSWs. With reference to potential inter-UCID, Estimated Active Intervals of activities play a very important role; therefore a change on project schedule may help overcome this error.

Let's have an example. Figure 7 describes an example of relationships between UML artifacts created in different phases of a software development process. If we change UML Artifact 1, we need to change UML Artifacts 4, 5, 8, 9 because of the relationships between them. Similarly, if we change UML Artifact 2, we need to change UML Artifacts 5, 6, 9, 10, 11. By tracing the relationships starting from UML Artifact 1 and UML Artifact 2, we can create two CSWs to respond to change requirements on UML Artifact 1 and UML Artifact 2 respectively (Figure 8). Based on the generated workflows, project manager can conduct other steps in project time management such as estimating activity resources, estimating activity durations, etc. Information about activity duration is used to detect potential UCIDs.

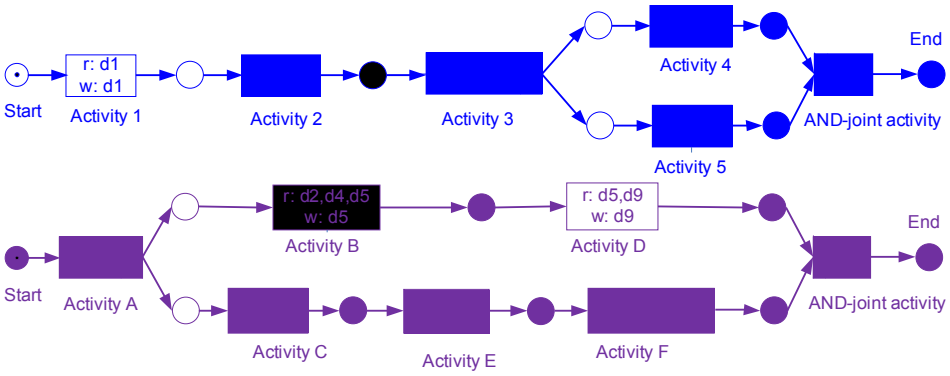


Fig. 8. Example of CSWs created based on the relationships between UML Artifacts

In Table 1, the minimum and maximum execution durations of each activity in CSWs described in Figure 8 are calculated from the Activity Duration Estimate, quantitative assessment of the likely number of work periods that will be required to complete an activity [18], of the corresponding activity in the project time management. Based on these values and the start time of the corresponding workflow, we can calculate the Estimated Active Intervals according to the formulas given in Section 5.1. After using the Inter-UCID detection algorithms, the following potential Inter-UCIDs are

reported: WW Inter-UCID between activity 3 and activity B on artifact 5, WW Inter-UCID between activity 5 and activity D on artifact 9, RW Inter-UCID between activity 3 and activity A on artifact 2, RW Inter-UCID between activity 5 and activity B on artifact 5. By applying the second Inter-UCID resolution method, modifying workflow structure, we get the synthesis CSW as described in Figure 9.

Because detecting potential UCIDs at build time is limited to workflows in which Estimated Active Intervals can be given before execution, solving this problem at runtime will be our next step. The model versioning system AMOR [21] offers some methods to resolve collaborative conflict in model versioning. Regarding this approach, all people who performed the changes are involved in eliminating the conflicts to obtain one consistent model version. We will consider applying this approach in our environment to increase the flexibility of the system.

Table 1**.** Time aspect of activities in CSWs described in Figure 7

CSW ID	**Start time P_w**	**Activity Name**	**Activity Duration Estimates (days)**	**Minimum and Maximum execution duration**	**Estimated Active Interval**
W1	5	Activity 1	7.5 ± 0.5	{7,8}	[5,13]
		Activity 2	5.5 ± 0.5	{5,6}	[12,19]
		Activity 3	11 ± 1	{10,12}	[17,31]
		Activity 4	6 ± 1	{5,7}	[27,38]
		Activity 5	7 ± 1	{6,8}	[27,39]
		AND-joint	0	{0,0}	[33,39]
W2	15	Activity A	6 ± 1	{5,7}	[15,22]
		Activity B	5 ± 1	{4,6}	[20,28]
		Activity C	5 ± 1	{4,6}	[20,28]
		Activity D	10 ± 1	{9,11}	[24,39]
		Activity E	5.5 ± 0.5	{5,6}	[24,34]
		Activity F	6 ± 1	{5,7}	[29,41]
		AND-joint	0	{0,0}	[34,41]

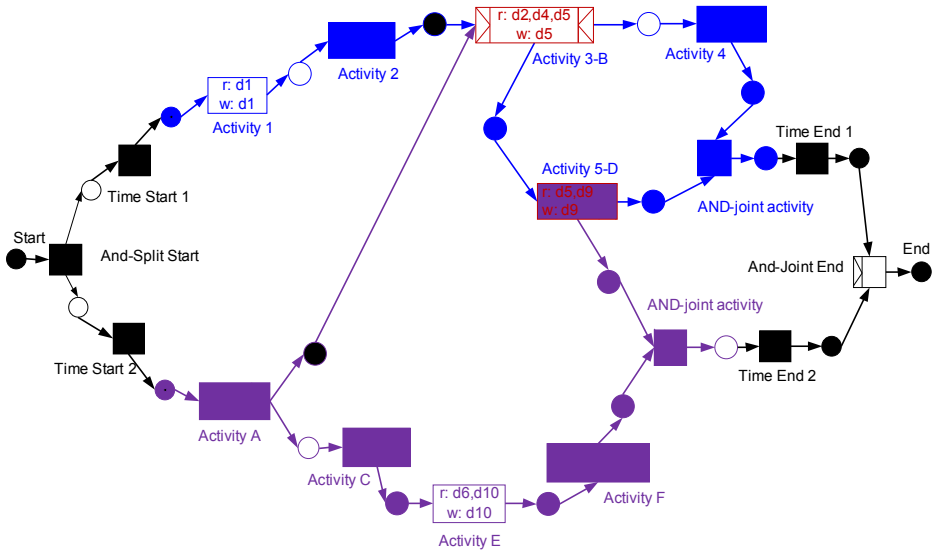


Fig. 9. Modified CSW with potential UCID corrected

8 Related Work

Workflow verification has attracted a lot of attention, especially control flow aspect. However, little research has been carried out on data verification in the workflow literature.

Reference [3] was one of the first studies to mention the importance of data-flow verification, and identified possible errors in the data-flow, like missing data, redundant data, conflict data, etc. Some general discussions on data flow modeling, specifications and verifications have been given, but without any detailed solution. The authors in [12] used data flow matrix and UML activity diagram to specify data flow. Based on this specification, an algorithm for detection of some data anomalies, such as missing data, redundant data, and potential data conflicts, was given [3]. In [11], a new workflow model, named Dual Workflow Nets, was defined to explicitly describe both control flow and data flow. A graph traversal approach was used in [10] to build an algorithm for detecting lost data, missing data and redundant data. More data flow errors were recognized and conceptualized as data flow anti-patterns and expressed in terms of temporal logic CTL* [5, 6]. By using temporal logic, available model checking techniques can be applied to discover these anti-patterns.

Nevertheless, all of these studies consider data flow errors in a single workflow only and no error removal method is given at all. In contrast to previous work, we address not only the interactions of concurrent activities inside a single workflow, but also the mutual influences between concurrent workflows, which are the sources of

data flow errors. In [19], we focused on identifying UCID situations and defining a new workflow model as an extension of Petri Nets. Two algorithms for detecting intra-UCID and inter-UCID were also given in this work. However, there are still many unsolved problems in [19] and this paper is its refined and extended version. In this paper, TDW is defined as an extension of Workflow Nets (WF-Nets) instead of Petri Nets. Because the two algorithms in [19] had many common steps, if we use them separately, execution cost would be high. Therefore, these two algorithms are combined to reduce the cost and to form a more accurate and useful algorithm. Algorithm evaluation is also included in this version. Besides, some heuristics are provided to make the algorithm more flexible and effective. After that, some UCID resolution methods are proposed to help remove UCID errors. Finally, building a change support environment for cooperative software development is introduced as an application domain for our work.

Concerning the mutual influences of the concurrent workflows approach, the research closest to us is [7]. However [7] addressed the verification of workflow resource constraints, and in this work, by nature, handling the resource problem is simpler than the data problem. A Time Constraint Workflow Net was defined to model workflow. Then, they identified the problem of resource constraints in WFMS and proposed a pseudocode algorithm which checked the resource dependency between every two activities. Reference [4] used hybrid automata to model the influences between concurrent workflows, and adopted a model checking technique to detect resource conflict problems.

9 Conclusion and Future Work

In this paper, we have presented ***Unintentional Change in In-use Data*** *(UCID)* concept and classified types of UCID which can occur, between activities in a single workflow or in different concurrent workflows. We have also proposed a Time Data Workflow based on the WF-Nets with many attributes supporting UCID estimation. An algorithm which helps detect intra/inter-UCIDs in a Concurrent TDW Management System has been developed too. After that, algorithms evaluation and some solutions to resolve UCID problem are given. Finally, we have introduced a concrete project supporting software change development process in a cooperative software environment as an application using UCID theory to verify change processes at build time.

As future work, we will implement a prototype of Concurrent TDW Management System and evaluate the effectiveness of UCID detection algorithm by runtime analysis. Then, we will improve inter-UCID resolutions and refine the generated TDW after applying UCID resolution methods in the Concurrent TDW Management System. Detecting and correcting UCID at runtime are our next targets. We also plan to investigate formal verification methods to verify the correctness of our model and method. Finally, we will integrate our system into the open source WoPeD [17]. Another direction of our research is to extend the TDW and improve UCID detection algorithms to address errors in resource and access control constraints.

References

1. Lee, M., Han, D., Shim, J.: Set-based access conflicts analysis of concurrent workflow definition. In: Proceedings of Third International Symposium on Cooperative Database Systems and Applications, pp. 189--196. Beijing, China (2001)
2. Li, H., Yang, Y., and Chen, T. Y.: Resource constraints analysis of workflow specifications. J. Syst. Softw. 73, 2, pp. 271--285 (2004)
3. Sadiq, S., M. Orlowska, W. Sadiq and C. Foulger.: Data flow and validation in workflow modeling. In: Proceedings of 15th Australasian Database Conference. LI, H. pp. 207--214 (2004)
4. Kikuchi S., Tsuchiya S., Adachi M., and Katsuyama T.: Constraint Verification for Concurrent System Management Workflows Sharing Resources. In: Third International Conference on Autonomic and Autonomous Systems (2007)
5. Trčka N., van der Aalst W.M.P., and Sidorova N.: Analyzing Control-Flow and Data-Flow in Workfow Processes in a Unified Way. Technical Report CS 08/31, Eindhoven University of Technology (2008)
6. Trčka N., van der Aalst W.M.P., and Sidorova N.: Data-Flow Anti- Patterns: Discovering Data-Flow Errors in Workflows. In: 21st International Conference on Advanced Information Systems (CAiSE'09). LNCS, vol. 5565, pp. 425--439. Springer-Verlag Berlin Heidelberg (2009)
7. Zhong, J. and Song, B.: Verification of resource constraints for concurrent workflows. In: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 253--261 (2005)
8. Wil van der Aalst, Kees Max van Hee: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, MA (2004)
9. Zeng, Q., Wang, H. and Xu, D: Conflict detection and resolution for workflows constrained by resources and non-determined duration. Journal of Systems and Software 81(9), pp 1491--1504 (2008)
10. Sundari M.H., Sen A.K., and Bagchi A.: Detecting Data Flow Errors in Work-flows: A Systematic Graph Traversal Approach. In: 17th Workshop on Information Technology & Systems (WITS-2007). Montreal (2007)
11. Fan S., Dou W.C., and Chen J.: Dual Workflow Nets: Mixed Control/Data-Flow Representation for Workflow Modeling and Verification. In: Advances in Web and Network Technologies, and Information Management (APWeb/WAIM 2007Workshops), LNCS, vol. 4537, pp 433--444. Springer-Verlag, Berlin (2007)
12. Sun S.X., Zhao J.L., Nunamaker J.F., and Liu Sheng O.R.: Formulating the Data Flow Perspective for Business Process Management. Information Systems Research, 17(4), pp 374--391 (2006)
13. Heinlein, C.: Workflow and process synchronization with interaction expressions and graphs. In: Proceedings of the 17th International Conference on Data Engineering (ICDE '01), pp. 243–252 (2001)
14. Workflow Patterns, http://www.workflowpatterns.com
15. Russell N., van der Aalst W.M.P., and ter Hofstede A.H.M.: Designing a Workflow System Using Coloured Petri Nets. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) III, 5800, pp 1--24 (2009)
16. Awad, A., Decker, G. and Lohmann, N.: Diagnosing and Repairing Data Anomalies in Process Models. In: 5th International Workshop on Business Process Design. LNBIP, pp

1--24. Springer, Heidelberg (2009)
17. Workflow Petri Net Designer, http://193.196.7.195:8080/woped
18. PMBOK Guide Fourth Edition. Project Management Institute (2008)
19. Phan Thi Thanh Huyen and Koichiro Ochimizu: Detection of Unintentional Change on In-use Data for Concurrent Workflows. In: Proceedings of the 2010 International Conference on Software Engineering Research and Practice (SERP 10). Las Vegas, Nevada, USA (2010)
20. Elmasri, R. and Navathe, S. B.: Fundamentals of database systems, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA (1989)
21. Adaptable Model Versioning, http://modelversioning.org/
22. Masayuki Kotani and Koichiro Ochimizu: Automatic Generation of Dependency Relationships between UML Elements for Change Impact Analysis. Journal of Information Processing Society of Japan, vol. 49, no.7, pp 2265—2291 (2008)

Automata and Petri Net Models for Visualizing and Analyzing Complex Questionnaires – A Case Study –

Heiko Rölke

Deutsches Institut für Internationale Pädagogische Forschung
(German Institute for International Educational Research)
Solmsstraße 75, 60486 Frankfurt, Germany
roelke@dipf.de

Abstract. Questionnaires for complex studies can grow to considerable sizes. Several hundred questions are not uncommon. In addition, routings are used to distinguish between question paths for different respondents. This leads to the question of how to ensure validity and other important properties.
We examine this question for a case with even more demanding side conditions: An important part of the OECD study "PIAAC" (Programme for the International Assessment of Adult Competencies) is a background questionnaire (BQ) containing more than 400 questions. This BQ has to be adapted by all participating countries. Nevertheless, integrity of the overall system has to be ensured.

Keywords: automata, Petri nets, questionnaires, analysis

1 Motivation and Overview

Large scale studies in psychology, sociology, and for many other purposes try to find out characteristics of complete populations or at least of big parts of a population. International studies often aim at comparing the complete population of one country to that of another country. The well-known PISA study of the OECD, as an example, aims at comparing *all* students of age 15 worldwide. This is done by examining representative samples in each country that participates in PISA.

To be able to compare one first has to find out some background information about the people that are compared. This is mainly done by asking those people questions. Larger chunks of questions grouped together in order to find about the *background* of the surveyed people are called *background questionnaires*, or BQ in short.

The author of this paper was involved in the definition, implementation, national adaptation, and deployment of the BQ for the OECD PIAAC study.[1] The OECD is the "Organization for Economic Co-Operation and Development", see [5] for details. Among many other activities, the OECD is well-known for organizing world-wide comparability studies, like the PISA study. PISA [6] is the abbreviation of "Programme for International Student Assessment". The PIAAC study, "Programme for the International Assessment of Adult Competencies" can be seen as an extension of the PISA study for adults. It aims at finding out about skills needed by adults in order to be successful in everyday work life. See [7] for details about the PIAAC study. PIAAC is carried out by 24 countries all over the world (participating countries are located in North and South America, Europe, Asia and Oceania).

1.1 Background Questionnaire Properties

There is no exact definition of what a BQ is. It is therefore not possible to exactly determine properties that have to be valid for each and every BQ. Naively, it is just a bunch of questions that an interviewer has to present to an interviewee. In practice, in discussions with psychologists, sociologists, or other questionnaire practitioners, certain universally agreed principles and best practices become clear. From this starting point, desirable properties can be derived. Nevertheless, it is not possible in the moment to definitely define and answer all related questions. We strive for more general validity, though.

A BQ usually has one single entry or starting point, the first *item* or *question*. In practice, this is often a hidden item, where predefined data is imported. An example: One often knows the name of the interviewee in advance. Within the BQ, there may be many different paths through the question pool, often depending on previously entered data or chosen randomly. An item that is intended to be the last question of a BQ is called an *end item*. Again, this may be a visible item (a question) or a hidden item not visible to the interviewer. While there often only is one end item, for example thanking the interviewee for time and patience or, more technically, exporting the assembled data, this is not a standard requirement of a BQ.

Due to practical considerations, there often is the possibility to pause an interview or to break it off. While pausing has no implications for the structure, a break-off means that any item can be an end item or has a connection to an end item.

The normal flow through a BQ should not result in a dead end. A dead end is an item that was not considered to be an end item. Other requirements are more on the semantic side. Each possible question sequence has to make sense semantically. On the other hand, each desired or planned sequence has to be possible, e.g. by entering appropriate answers.

[1] The work was done in the international consortium responsible for implementing, deploying, and analyzing the study, led by ETS [3] in Princeton, USA. Most of the implementation work on the BQ was done by the CRP Henry Tudor [2] in Luxembourg.

1.2 Overview

The rest of the paper is structured as follows: In Section 2 we give an overview on the BQ of the PIAAC study. We also give examples of the format in that the BQ is defined. Following up on that we develop first simple models for the PIAAC BQ in Section 3. The models are put into practice in Section 4. They are used to gain quite some insight and find errors, but are not sufficiently powerful to represent all important aspects of our application. So we carry on in Section 5 with more powerful Petri net models that allow for more sophisticated analysis. We conclude in Section 6 with an outlook on further work and possible generalizations.

2 The PIAAC Background Questionnaire

The PIAAC study mainly consists of two important parts: a background questionnaire (BQ) and cognitive tests (cognitive items, CI). Both parts are embedded into an overall workflow that controls all parts of the survey. This workflow is implemented in the same way as the BQ.

The PIAAC BQ starts with general questions about the interviewee to find out whether he is suited to take part in the survey or not. Afterwards questions in different categories are asked, grouped together in blocks. Examples for such blocks are questions about the educational background, skills needed in everyday work, and questions about private life related to work skills. In order to shorten the overall interview time, parts of the blocks are arranged in a rotated design so that not all interviewees are asked the same questions. Another example of inter-block routing is that certain blocks are not administered if the requirements for asking these questions are not fulfilled, e.g. questions about current work in case of an unemployed interviewee.

In addition to the inter-block routing, complex routing is used within blocks to administer the right questions. A good example for such a routing are questions about the education of an interviewee: If an interviewee has never been to an university it is useless to ask questions about academic degrees. Respective questions should be skipped. Another typical situation includes loops: One might be interested in the degree of skills related to foreign languages. To accommodate speakers fluent in multiple languages, some kind of cycle or loop is needed.

The code example in Figure 1 shows an example of a questionnaire item with a free text entry. The XML syntax is not important here.[2] The item group that is defined in the code snipped defines a single item, i.e. one question is administered. The item has a unique identifier (ID), instruction text and answering possibilities. In this case a free text entry of length 12.

The second code example in Figure 2 shows a routing with two possible targets. This is a hidden item, i.e. an item that is not displayed but used internally.

[2] The XML syntax of the PIAAC BQ has been specially designed for this purpose. At the time of writing of this paper only limited support like editors or visualizers is available.

```
<itemGroup id="CI_PERSID" responseCondition="ALL" layout="list">
  <item id="CI_PERSID">
    <instruction>Please enter the sampled person ID</instruction>
    <responses layout="radioButton">
      <response code="00" freeTextEntry="true"
         freeTextEntrySize="12" > Sampled Person ID:[FTE]</response>
    </responses>
  </item>
</itemGroup>
```

Fig. 1. BQ code example: free text entry

```
<itemGroup id="CI_skip-C-200Rule" layout="list"
   responseCondition="ALL" hidden="true">
<item id="CI_skip-C-200Rule"/>
<routing>
  <condition>
    <operator type="equal">
      <variable name="CI200Rule"/>
      <constant>NI</constant>
    </operator>
  </condition>
  <then>
    <goto itemGroup="CI200Rule"/>
  </then>
  <else>
    <goto itemGroup="CI_start"/>
  </else>
</routing>
</itemGroup>
```

Fig. 2. BQ code example - conditional routing

The routing is conditional. It is based on the value of the variable `CI200Rule`. Based on this variable, the BQ jumps to item `CI200Rule` or `CI_start`.

Each variable can only be written once. There is a one-to-one relationship between an item and a variable. The variable has the same name as the item where it is initialized and written. Afterwards the variable can only be read, not changed or deleted. There is a notable exception to this rule: It is possible to go back in the questionnaire, for example in case of an error noticed later on. If this is done, the variables connected to the items eventually asked again can also be written again.

The PIAAC BQ together with the overall survey workflow contains more than 600 items. It has one single start item and one single end item. It is possible to break-off the interview at many items, but not all. Break-off leads to a special item that asks for the reason for the break-off.

3 BQ Modeling

A basic modeling strategy for background questionnaires is relatively straightforward. Items (and/or item groups) can be modeled for example as states of a finite automata. Going from one question to the other is a matter of transition from one state the the next. Routing can be modeled as conflicting state transitions. We will now have a closer look at this idea and discuss whether it is sufficient below.

3.1 Automata models

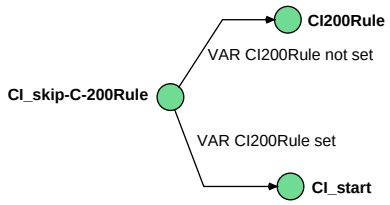


Fig. 3. Finite automata for code in Fig. 2

Figure 3 illustrates the idea of an automata model for the BQ. The automata implements the BQ code example of Figure 2. State `CI_skip-C-200Rule` is connected to the states `CI200Rule` and `CI_start`. The actual transition depends on the variable `CI200Rule` as described above. This data dependency causes problems with this basic model. We will come back to this problem later on.

The benefit even of such a simple formal model is twofold: Once a questionnaire is transformed to a finite automaton, automatic as well as manual

inspection is possible. Automatic inspection can check important properties like connectedness and reachability of the final state(s). Manual inspection is enabled by using a graphical tool that displays the complete BQ. This allows for a much more convenient way of getting an overview of the BQ. It is nearly impossible to follow all routings in the sequential XML format, even if this is supported by an appropriate style sheet (XSLT, see [19]) using links and an overview frame. See Figure 8 to get an idea of the complexity of the BQ. Note that this figure only shows a small part of the overall questionnaire. The model shown in the figure is implemented as a Petri net, not an automaton.

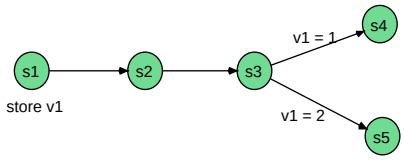


Fig. 4. Data dependent routing

Figure 4 illustrates a general problem with the simple modeling approach. In state `s1` the variable `v1` is written. Afterwards another state (`s2`) is reached and then `s3`. Only in this state the value of `v1` is read again to determine which state (`s4` or `s5`) should be reached next. This means that the variable is used non-locally. While such a situation is common in questionnaires, it is not possible to model a non-local usage of a variable in an ordinary finite automata.[3]

3.2 Petri net models

To overcome the problem of non-local variable usage illustrated above, we remodel the very same BQ part as a Petri net.[4] This can be seen in Figure 5.

As we can see in the figure, the problem can easily be overcome. Items, previously modeled as states in the automaton, are now modeled as places of the Petri net. Transitions have been introduced between the states/places. The places can be seen as the static part, e.g. question or instruction. The transitions are the dynamic part, e.g. the answer given to the respective question and/or the stored variable. Depending on the values stored and retrieved in the variables, the resulting net can be a Place-/Transition net or a colored Petri net.

Place-/Transition nets are possible for variables with restricted (=finite) domains. Luckily, this type is most commonly used in BQs. The vast majority of

[3] This is not completely true, because for variables with finite domains it would be possible to enumerate all reachable states for all values of all variables. Nevertheless, such a model would be hard to read and not very useful.

[4] Petri nets are not an arbitrary choice. They offer various advantages: graphical representation, formal analysis, tool support.

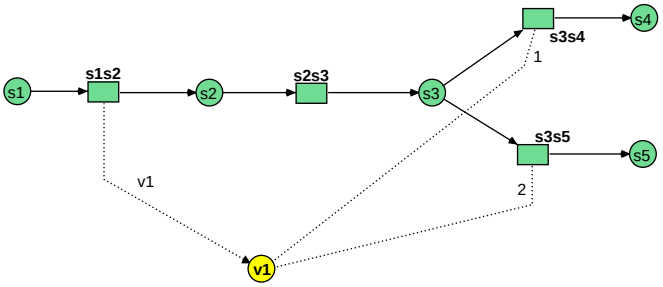


Fig. 5. Data dependent routing of Fig. 4 as PN model

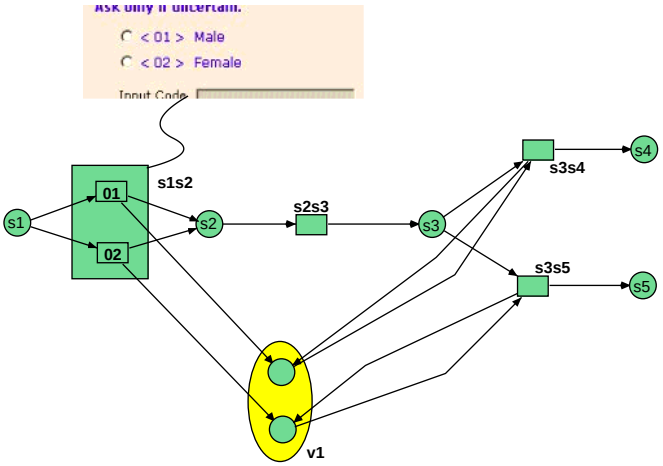


Fig. 6. Refinement of Fig. 4 as PT net

questions is of a single or multiple-choice type. Such a question is illustrated in Figure 6. The first question in this example is about the gender of the interviewee. Only two answers are possible. This can be modeled by a refinement of the net of Figure 5, as shown above. The resulting net is a P/T-net and can be analyzed using the respective tools.

For free text variables or numbers such a modeling would not be possible. Instead, we can use colored nets that support high-level data structures for places and variables directly. While such models are more difficult to analyze they offer other advantages. We will stick to P/T-nets for the moment and come back to the advanced net models later on in Section 5.

4 Modeling and Analysis for PIAAC

The definition, implementation, national adaptation, and deployment of the PIAAC BQ was driven by a high time pressure. Pre-existing questionnaire parts had to be combined and extended. A compromise had to be found that was (a) not too long, (b) implementable world-wide - both a cultural and a political challenge, and (c) able to gather enough data to give answers to the grounding questions of PIAAC. Therefore the work on the BQ started using a semi-structured approach (printable and human-readable Excel sheets), to be able to quickly disseminate all intermediate versions and get feedback. Only late in the process, this was transformed to a well-defined XML format. Therefore also the work on the formal analysis of the BQ started late and is not completely done yet.

The first attempt to get some insight into the BQ structure handled the BQ as a graph. Only an internal model was built, without any graphical representation. Variables were neglected, only the control flow was mapped. From this simple model some important insights were possible: We found dangling routings (jumps to undefined items, e.g. due to spelling mistakes), duplicate item names and isolated nodes - items that could never be reached. On the other hand, it turned out to be quite tedious to verify and analyze the error reports of the first analyzer because of the lack of a graphical representation.

Therefore we implemented another approach targeting on Petri nets. This formalism was chosen to be able to benefit from the advanced set of tools available, allowing to visually inspect a net and formally analyse it at the same time. Our work greatly benefited from the existence and widespread support of the PNML standard - see [8] for an overview or the web site [16] for more information. The usage of PNML allowed to implement the modeling process – modeling a Petri net that represents a specific BQ – as an XSLT transformation.

The first attempt to do so replicated the graph analyzer mentioned above. Variables were neglected, all routing possibilities were handled equally without interpreting the routing conditions. This resulted in a PNML net definition file only containing places, transitions, and arcs. An example can be found in Figure 7. Note that `[...]` means the omitting of plenty of PNML code.

```
<?xml version="1.0" encoding="UTF-8"?>
<pnml>
  <net id="piaac-BQ-DE-001" type="piaac-analyse">
[...]
   <place id="B_C02b1DE2b">
      <name>
        <text>B_C02b1DE2b</text>
      </name>
    </place>
[...]
    <transition id="t_B_C02b1DE2b_B_Q02b2DE2_32">
      <name>
        <text>t_B_C02b1DE2b_B_Q02b2DE2_32</text>
      </name>
    </transition>
[...]
    <arc id="B_C02b1DE2b_t_B_C02b1DE2b" source="B_C02b1DE2b"
       target="t_B_C02b1DE2b">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
[...]
  </net>
</pnml>
```

Fig. 7. Example PNML Code

Our modeling approach does not generate any graphical information. Therefore a tool had to be found that is able to import PNML files, construct a graphical representation automatically, and analyse the P/T-net. We chose the ProM tool for this purpose. For more information on ProM, see [18] and [17]. ProM especially well supports the handling of large nets and arranges the net elements in a way that is very well readable for the human eye.

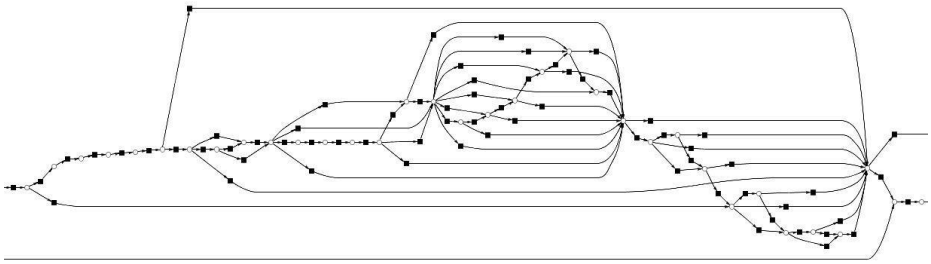


Fig. 8. BQ part as a P/T-net

In Figure 8 a small part of the complete BQ net is shown. As said before this is a simple model in the sense that the variables have been omitted. The figure is presented here just for illustration purposes. It serves to get an impression of the complexity of the overall BQ model. The complete model is way too big to be presented here. The layout of the example net has been done automatically by ProM.

Once available as a P/T-net in ProM, the build-in analysis means can be used. The PIAAC BQ has a single start item and a single end item. All items should be reachable and there may not be a dead end. The resulting BQ net therefore has to be a net with workflow properties. This is easily analyzable in ProM and gives good insight into the BQ definition. Doing so, we were able to find all the error types mentioned above with the big advantage of directly *seeing* the problems in the net graph. It now is way simpler to find fixes for the errors.

5 Advanced Net Models

In this section, we discuss experimental models that have not been used so far for the complete BQ. Nevertheless, as this is ongoing work, this will change soon.

To get deeper insight into the formal properties of a BQ, factoring in the variables and (routing) conditions is necessary. However, this may lead to way more complicated models, as we can see from a simple example. For this, we extend the example of Figure 6 to four possible answer categories on item `s1`, two answer categories on item `s2` and three conditional routings after `s3` relying on the variables `v1` and `v2`:

- `s4, if v1 = 1 and v2 = 1`
- `s5, if v1 = 2`
- `s6, if v1 > 2 or v2 = 2`

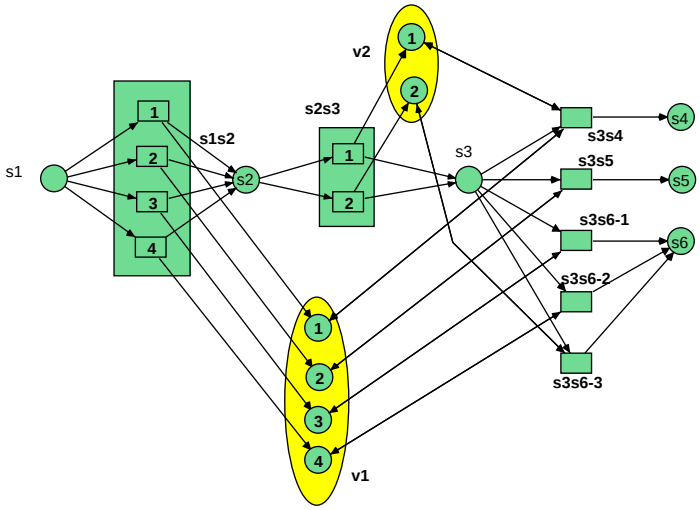


Fig. 9. Additional net elements for variables and conditions

Even this mild extension leads to a more complicated net model. Especially the last routing condition (leading to `s6`) is interesting, as it has to be unfolded to three transitions: two for the "greater-than" and and extra for the "or"-part. In real settings, this can easily grow to huge amounts of transition. As an example, in the German BQ there are routing conditions combining 5 variables, each enclosing up to 16 possible values, to route to more than 10 targets.

Another possibility is to model colored Petri nets instead of P/T-nets. Colored Petri nets directly support high-level data structures and variables. Nevertheless, they still allow for analysis, the necessary unfolding process is done inside the tool, for example the CPN Tools [4,10]. This option is under investigation.

In the moment, the PIAAC BQ is defined by means of writing XML code. This is tedious and error-prone work. The direct syntax can be checked relatively easily, but syntactical errors like missing routing targets are harder to detect. Semantic errors like dead ends even harder. Parts of these problems could be overcome by using a high-level Petri net formalism like Workflow Nets [9,15] as a means for rapid prototyping and/or adding small changes and corrections. Workflow Nets are directly executable, so that changes can be tried out easily. The graphical modeling permits typical errors mentioned above and gives a good overview on what one is doing. To support large BQ models, means for abstraction and rapid modeling are necessary. Workflow Nets offer such means.

Abstraction is possible in form of object tokens of the underlying reference net formalism [12,13,14] and by dynamic transition refinement [11]. Rapid modeling is facilitated by workflow patterns, a special form of net components - see [1] for an overview.

6 Conclusions and Outlook

We found a way of making use of well-known and well-understood formalisms and tools for a new domain. While some good results have already been achieved, several ways of extending the work are possible:

- The BQ analysis should be integrated into the normal BQ definition and release process. In the moment, it still requires manual work. It has been done completely only for the German version of the BQ.
- While the single steps of the analysis approach are rather straightforward, the combination still requires some manual work. This should be simplified to allow non-expert users to do the analysis on their own.
- The advanced models of Section 5 can be used directly for analysis of the BQ. This is still in an experimental state. We try to partition the BQ net into independent sub-nets to circumvent the net size explosion.
- In the moment, the BQ definition and especially the national adaptation process has long turn-around times. Countries request changes without the possibility to try them out beforehand. Using the rapid prototyping idea of Section 5 they could first try out the changes themselves and only request approval afterwards, once the changes are stable and working on the national level.

The examples and the analysis shown in this paper could partly be modeled using a sequential modeling formalism. However, Petri nets offer big advantages when it comes to non-local dependencies. For example, in the PIAAC BQ, some of the questions should only be asked a limited number of times in a country. This is straightforward to model in PN but maybe more difficult in other modeling formalisms.

The analysis of background questionnaires could benefit a lot from a sound formalization of what a BQ is. As mentioned early in the paper, no such definition exists so far. This question needs further research. Especially the similarity of BQs and workflows should be analyzed more deeply.

References

1. Lawrence Cabac. Net components: Concepts, tool, praxis. In Daniel Moldt, editor, *Petri Nets and Software Engineering, International Workshop, PNSE'09. Proceedings*, Technical Reports Université Paris 13, pages 17–33, 99, avenue Jean-Baptiste Clément, 93 430 Villetaneuse, June 2009. Université Paris 13.
2. Centre Research Public Henri Tudor (CRP-HT). http://www.tudor.lu. WWW.
3. Educational Testing Service (ETS). http://www.ets.org. WWW.

4. Computer Tool for Coloured Petri Nets (CPN Tools). http://wiki.daimi.au.dk/cpntools/cpntools.wiki. WWW.
5. Organization for Economic Co-Operation and Development (OECD). http://www.oecd.org. WWW.
6. Programme for International Student Assessment (PISA). http://www.pisa.oecd.org. WWW.
7. Programme for the International Assessment of Adult Competencies (PIAAC). www.oecd.org/els/employment/piaac. WWW.
8. L.M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. A primer on the petri net markup language and iso/iec 15909-2. *Petri Net Newsletter*, 2010.
9. Thomas Jacob, Olaf Kummer, Daniel Moldt, and Ulrich Ultes-Nitsche. Implementation of workflow systems using reference nets – security and operability aspects. In Kurt Jensen, editor, *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, August 2002. University of Aarhus, Department of Computer Science. DAIMI PB: Aarhus, Denmark, August 28–30, number 560.
10. Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, Volume 9(3):213–254, 2007.
11. Michael Köhler and Heiko Rölke. Dynamic transition refinement. *Electronic Notes in Theoretical Computer Science*, 175:119–134, June 2007.
12. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
13. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. Renew – the Reference Net Workshop. Available at: http://www.renew.de/, August 2009. Release 2.2.
14. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. *Renew – User Guide*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 2.2 edition, August 2009. Available at: http://www.renew.de/.
15. Daniel Moldt and Heiko Rölke. Pattern based workflow design using reference nets. In Wil van der Aalst, Arthur ter Hofstede, and Mathias Weske, editors, *Proceedings of International Conference on Business Process Management, Eindhoven, NL*, volume 2678, pages 246–260, 2003.
16. Petri Net Markup Language (PNML). http://www.pnml.org/. WWW.
17. Process Mining Toolkit (ProM). http://prom.win.tue.nl/tools/prom/. WWW.
18. W.M.P. van der Aalst, B.F. van Dongen, C. Günther, A. Rozinat, H. M. W. Verbeek, and A. J. M. M. Weijters. Prom: The process mining toolkit. In *Proceedings of the BPM 2009 Demonstration Track, Volume 489 of CEUR-WS.org, Ulm, Germany*, 2009.
19. XSL Transformations (XSLT). http://www.w3.org/tr/xslt. WWW.

Deadlock Control Software for Tow Automated Guided Vehicles using Petri Nets

Carlos Rovetto[1], Elia Cano[1] and José-Manuel Colom[2]

[1] Dpt. of Computer Science and Systems Engineering (DIIS)
[2] Aragón Institute of Engineering Research (I3A)
University of Zaragoza, Spain
{carlos.rovetto,elia.cano}@utp.ac.pa, jm@unizar.es

Abstract. Factoring and warehouse distribution centers face numerous and interrelated challenges in their efforts to move products and materials through their facilities. New technologies in navigation and guidance allow true autonomy with more flexibility and resource efficiency. In this paper we investigate a complete design approach to obtain deadlock-free minimal adaptive routing algorithms for these systems. The approach is based in an abstract view of the system as a Resource Allocation System. The interconnection network and the routing algorithm elaborated by the designer, are the initial information used to obtain in an automatic way a Petri Net model. For this kind of routing algorithms, we prove that the obtained Petri Net belongs to the well-known class of S^4PR net systems, and therefore the rich set of analysis and synthesis results can be applied to enforce the liveness property of the routing algorithm.

Key words: *AGVs, Control Software, Resource Allocation Systems (RAS), Modular Models, Structural Analysis.*

1 Introduction

Nowadays, many factories and warehouse and distribution centers use *Automatic Guided Vehicles* ($AGVs$) for item transportation among workstations. The wheeled trailers are the most productive form of AGV for tugging and towing because they haul more *conveyor-loads* per trip than other AGV types. In this paper we consider a warehouse distribution center as a programmable system for conveyor-loads movement among workstations using tugger AGV. The problem to be investigated concerns the design of *Deadlock-Free* minimal adaptive routing algorithms for the guidance system of tuggers $AGVs$, travelling into an warehouse distribution center. We say that the routing algorithm is minimal because only routes of minimal length between two workstations are taken into account. Moreover, the routing algorithms we are considering are adaptive in the sense that the route of a conveyor-load is constructed segment by segment. The assignment of a segment to the route of a conveyor-load is done in a workstation when the first trailer try to leave the workstation towards its destination workstation.

From the methodological point of view, the design of deadlock-free minimal adaptive routing algorithms is a complex task, where the designer experience is required because deadlock states can appear. There exist several approaches to cope with this problem [1–5]. They consider more general routing algorithms than those considered in this paper (including, for example, non-minimal routes). Because this generality, very few powerful analysis and synthesis results are available.

Our approach gives a full design cycle for *minimal adaptive routing algorithms* using Petri Nets as formal model that allows structural analysis of the liveness property of the model. Afterwards, if it is necessary, the initial routing algorithm is changed. From the point of view of software engineering, in the context of the control software for $AGVs$ systems, this paper intends to make contributions in the following directions: (a) The formalization of an *abstraction* process of the system to retain only the relevant characteristics in the study of deadlock problems in the routing software of $AGVs$. This abstraction is constructed around a minimal set of concepts – processes and resources. (b) The demonstration that for AGVs with minimal adaptive routing algorithms, the proposed abstraction process gives rise to models belonging to a well known class of Petri Nets named S^4PR, and so, we have many available results to cope with these systems. (c) *A modular methodology* to construct the models based on the specification of processes with resources that form the modules. The modules are composed by the fusion of common shared resources (segments) by different modules. This paper is organized as follows. In Section 2 an illustrative example is presented. In section 3 the proposed methodology is presented in detail. Section 4 presents the first step of the methodology consisting of the abstraction of the warehouse distribution center and the routing algorithm to retain only those aspects related to the appearing of deadlocks. Section 5 is devoted to the Petri Net model representing the Resource Allocation view of the system. This section also proves that the Petri Nets obtained for these routing algorithms belong to the class of the S^4PR nets. Section 6 presents the analysis and synthesis phases of the methodology that profit the theoretical results known for the class of S^4PR. Finally, section 7 presents some conclusions.

2 An Example

In this section, a simple example of a warehouse distribution center, will be presented. The specification of this example illustrates the typical situation in the transportation system of items. We start with a layout of the shipping areas defined by a set of workstations WS and a set of segments SG interconnecting the workstations. The connection pattern among workstations will be called the $framework$ of the warehouse distribution center. The example that we are considering is an unidirectional ring in clockwise fashion as underlying framework. There are four workstations WS=$\{w_0, w_1, w_2, w_3\}$ and they are interconnected by a set of segments SG=$\{sa_0, sa_1, sa_2, sl_1, sl_2, sl_3\}$. This warehouse distribution center is depicted in schematic way in Fig. 1.a. Observe that if a workstation

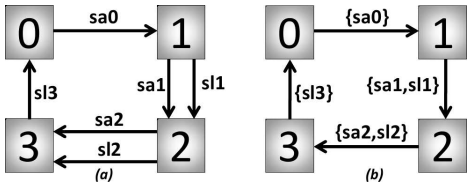


Fig. 1. a) Framework skeleton and its, b) Warehouse Graph.

has two output segments, a train can follow any of them. This decision is taken by the local *minimal adaptive routing algorithm* of the workstation. The other defining element of the warehouse distribution center is the behavior of the conveyors because a train can tow single or multiple trailers hence the length of the conveyors is variable. As the conveyors flow in pipeline fashion through the framework, these can have simultaneously allocated several segments of the framework. The first trailer of the *AVG* train is the **head** of the conveyors and reserve the segments to transit; the last trailer is the **tail** and release them. Traditionally, each segment supports only one conveyor at time to avoid collisions. In our example, each workstation executes, an instance of the following minimal adaptive routing algorithm parameterized by the identity of the workstation.

ALGORITHM 1 *Minimal Adaptive Routing Algorithm for workstation i.*
Input: The head trailer cl from the conveyor-load queue.
Local: $S_i \subseteq SG$*, output segments for workstation* i
$F \subseteq S_i$*, set of non-assigned output segments*
Output: The next segment to be used for cl
begin
if $(destination(cl) = i)$ **then** *store the conveyor-load cl in workstation i*
else
if ($sa_i \in F$) **then** *use* sa_i *;* $F{:=}F\backslash\{sa_i\}$
else
if $((destination(cl) < i) \wedge (sl_i \in F))$ **then**
use sl_i *;* $F{:=}F\backslash\{sl_i\}$
else *enqueue cl*
end if
end if
end if
end

That means the workstation knows its non-assigned output segments and the algorithm assigns, if it is possible, the output segment that the first trailer must follows in order to reach its destination. In other words, to reach a destination workstation, w_d, different to the current workstation w_i, the algorithm tries to assign the output segment sa_i if it is an output free segment of w_i. Otherwise, sl_i is assigned if this segment is an output free segment of w_i and the index d of the workstation w_d is less than the index i of w_i. This reservation is done by

the head trailers. The intermediate trailers follow through the reserved segments and the tail trailer release the segments that they will be added to the set of free segments F. The design of minimal adaptive routing algorithms can lead to solutions where deadlock states can be reached. A deadlock state, in a warehouse distribution center, arise when a set of conveyor-loads are in transit to their respective destination workstations but all of them are stopped forever in intermediate workstations. They are waiting for the availability of output segments of these intermediate workstations that have been previously assigned to conveyor-loads belonging to this set. Therefore, none of the implied conveyor-loads will reach their destination workstations. The minimal adaptive routing algorithm of our example presents this anomaly that we illustrate by means of the following deadlock state. We have four conveyor-loads, $\{cl_1, cl_2, cl_3, cl_4\}$, each one composed by more than one trailer. It is easy to verify that the state described in table 1, for the four conveyor-loads in transit, is reachable, where H and T represent the current workstations of the head and tail trailers, respectively. The rest of the columns in the table 1 represent: *Allocated segments*− segments assigned to the conveyor-load; *Destination workstation*− represents the destination workstation of the conveyor-load; *Next segment*− segment to be assigned to the head trailer according to the minimal adaptive routing algorithm. Observe that

Conveyor -loads	**Trailers**		**Allocated Segments**	**Destination Workstation**	**Next segment**
	H	T			
cl_1	w_0	w_3	sl_3	w_1	sa_0
cl_2	w_1	w_0	sa_0	w_2	sa_1
cl_3	w_2	w_1	sa_1	w_3	sa_2
cl_4	w_3	w_2	sa_2	w_1	sl_3

Table 1. Deadlock state reached in the example concerning four conveyor-loads.

all conveyor-loads are in intermediate workstations and in order to advance in the warehouse distribution center, all conveyor-loads need segments (those given by the minimal adaptive routing algorithm) that they are allocated by other conveyor-loads in the same set (compare the two columns "Allocated Segments" and "Next segment" in the table 1). On the other hand, none of the tail trailers can release segments because if some tail trailer moves ahead, it will be in the same workstation that the head trailer and this is not possible. Therefore, we have reached a deadlock state where the four classical necessary conditions for the existence of a deadlock are fulfilled. Finally, you can observe that although we are in a deadlock state, there exist two segments, sl_1 and sl_2, that they are free, and the minimal adaptive routing algorithm cannot assign these segments to the four conveyor-loads of our scenario.

3 The proposed methodology

In this paper we advocate for a methodology where, after an analysis phase of the model obtained from the framework (the interconnection network) and the minimal adaptive routing algorithm, a synthesis procedure transform the original routing algorithm to make it deadlock-free. In order to implement this methodology we will make use of Petri Net models. Therefore, the first task will be the construction of the Petri Net model that retains only those aspects related to the appearing of the deadlock states. Deadlocks appear as a consequence of the allocation of the segments by the conveyor-loads in transit in the warehouse distribution center. Therefore, we will adopt a *Resource Allocation* perspective to abstract the system *(RAS view of the warehouse distribution center)* where segments will be considered as resources, that they are used in a conservative way *(they are not created nor destroyed)* by the user processes that they are the conveyor-load moving from a source workstation to a destination workstation. In next section, from the framework and the routing algorithm we will obtain a *Routing Graph* for each destination workstation. One of these Routing Graphs represents a transition graph where we present the reachable states of a conveyor-load, composed by more than one trailer, from a source workstation of the warehouse distribution center to the destination workstation corresponding to the Routing Graph. From these Routing Graph and the segments considered as resources, in section 5 we obtain a Petri Net that, in the case of minimal adaptive routing algorithms, belongs to the well known class of S^4PR nets. Now, using the known analysis results for this class of nets we can characterize the existence of deadlocks using a structural reasoning. The synthesis procedure is based on the methods for liveness enforcing developed by different authors [6] [7] [8]. The Fig. 2 presents in graphical form the methodology we propose for the design of deadlock-free minimal adaptive routing algorithms. In this methodology, the Petri Nets play a central role, because they are used to model the *RAS* view of the warehouse distribution center, and this is the reason of this paper: to present how to obtain these Petri Nets and to prove that they belong to a previously known class of Petri Nets (S^4PR), and so well studied.

4 Construction of the Routing Graph

The goal of this section is to represent, step by step, the construction of the so called *Routing Graph* from the information about the framework of the warehouse distribution center and the minimal adaptive routing algorithm. This Routing Graph will represent the sequence of states that a conveyor-load must follow to reach a given destination workstation, w_i. The definition of the state concerns the set of segments that the conveyor-load is using each time. Therefore, *RG* give us the so called *Resource Allocation (RAS)* view of the warehouse distribution center. First, the framework is formalized through the *Warehouse Graph (WG)*. The WG is a labeled graph $WG{=}(W, S)$, where W is a set of vertices and S is a set of edges. The set W is equal to WS and $S{\subseteq}W{\times}2^{SG}{\times}W$, where

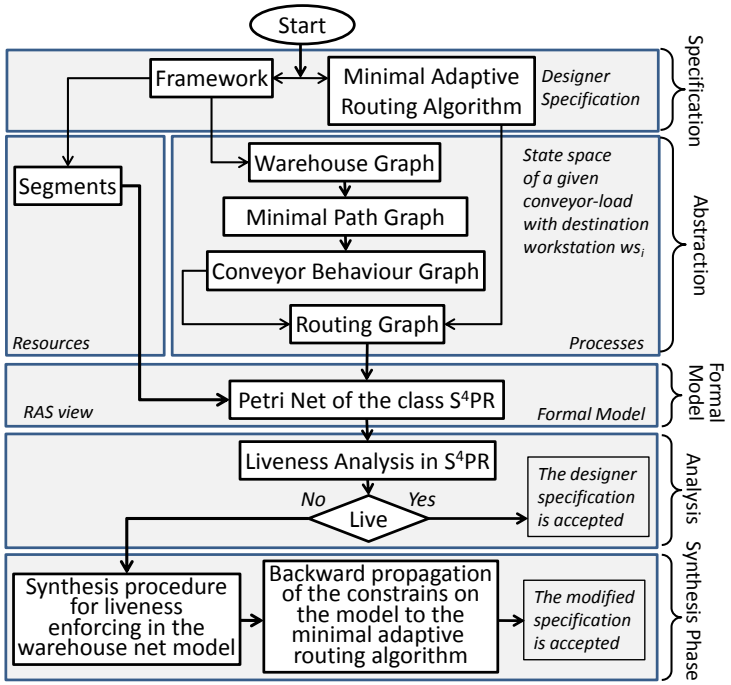


Fig. 2. Design Flow Methodology.

WS is the set of vertices and SG the set of segments. An edge $(w_1, s, w_2) \in S$ means that there exists a set of segments $s \subseteq SG$ from the workstation w_1 to the workstation w_2, as it is shown in the Fig. 1.b. We are considering the class of minimal adaptive routing algorithms. Therefore we will represent for each destination workstation w_i all the paths of minimal length from a workstation w_j to the destination workstation w_i. This information is captured into the *Minimal Path Graph (MPG_i)* with destination workstation w_i. Each one of these graphs is a subgraph of the $WG = (W, S)$ and it will be an acyclic directed labeled graph, $MPG_i = (V, E)$, where $V = W$, and $E \subseteq S$, verifying that:

1. All output arcs of w_i in WG do not belong to E.
2. The function $L_i : V \rightarrow \mathbb{N}$ is well defined: $L_i(w_i) = 0$ and $\forall w_j \neq w_i$, $L_i(w_j) = k$, where k is the length of the minimal path from w_j to w_i in the WG.
3. All arcs $(w_1, s, w_2) \in S$ in WG, such that $L_i(w_i) + 1 \neq L_i(w_2)$, do not belong to E.

The graphs MGP_i for the example of Fig. 1 are depicted in the Fig. 3. Observe that we will have four of these graphs, one for each possible destination workstation. Each MPG_i can be seen as the set of paths that can follow a conveyor-load originated in the workstation w_j with destination workstation w_i, an this path satisfy the minimality condition of the considered routing algorithm. Nevertheless, we are considering conveyor-loads with more than one trailer of length,

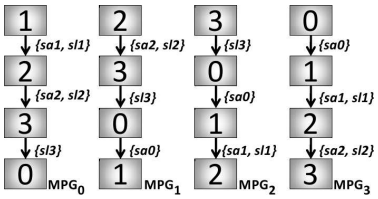


Fig. 3. Minimal Path Graph for all destination of our example.

because a conveyor-load with only one trailer cannot participate into a deadlock, since a deadlock must fulfill the *Hold and Wait* condition. Therefore in our model we must distinguish states according to the workstations where the head and tail trailers can be found. On the other hand, it is important to say that the advancement of the head trailer from a workstation to another can be done if and only if there exists at least a segment that can be allocated for this movement. Segments, therefore are resources in our *RAS* view of the warehouse distribution center. If the head trailer allocates the needed resources for the movement of the full conveyor-load, the tail trailer take charge of the release operation after the use of a segment. In order to represent the states of a conveyor-load with destination workstation w_i we will construct, from the MPG_i, the so called *Conveyor Behaviour Graph (CBG)* for the destination workstation w_i, CBG_i=(Q, F), verifying that.

1. $Q \subseteq V \times V$, where $\forall w_h, w_t \in Q$, $w_h = w_t$ or $L(w_h) < L(w_t)$. That is, the first component of the defined states corresponds to the workstation where the head trailer is, and the second to the workstation where the tail trailer can be found.
2. $F \subseteq Q \times \{A, R\} \times 2^{SG} \times Q$, where F will contain the following edges:
   (a) Allocation edges $((w_{h1}, w_t), A, S, (w_{h2}, w_t))$, $\forall\ w_t \in V, \forall((w_{h1}, s, w_{h2}) \in E$.
   (b) Release edges $((w_h, w_{t1}), R, S, (w_h, w_{t2}))$, $\forall\ w_h \in V, \forall((w_{t1}, s, w_{t2}) \in E$.

Obviously, CBG_i is a directed acyclic graph because MPG_i is also a directed acyclic graph. The Fig. 4 shows the Conveyor Behaviour Graph for destination workstation 0, CBG_0, corresponding to the MGP_0 of Fig. 3. Finally, to construct the announced Routing Graph of our warehouse distribution center we need to incorporate the information corresponding to the routing algorithm. The routing algorithm is a function $R{:}WS \times WS \rightarrow 2^{SG}$, such that if wc is the current workstation of the head trailer and wd the destination workstation of the conveyor-load $R((wc, wd))$ determines the output segments of wc to be allocated in order to reach the destination workstation. The model that we will construct is a possibilistic model, in the sense that from a current workstation we can have several alternative transitions, each one corresponding to a different allocated segment. Therefore in order to represent this information of the routing algorithm, from each CBG_i we will construct the so called *Routing Graph*

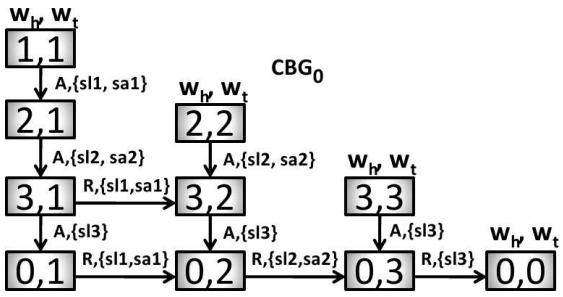


Fig. 4. Conveyor Behaviour Graph for destination workstation 0.

(RG) to the destination workstation w_i, RG_i=(Q', F'), where $Q \subseteq V \times V \times SG^*$ represents the set of states in which a conveyor-load can be found.

ALGORITHM 2 *Construction of the* $RG_i = (Q', F')$
Input: $CBG_i = (Q, F)$
Output: $RG_i = (Q', F')$
begin
next-level := $\{(w, w, \varepsilon) | (w, w) \in Q\}$
Q' := *next-level*; $F' := \emptyset$;
while *next-level* $\neq \emptyset$ **do**
current-level := *next-level*; *next-level* := $\emptyset$;
for each $(w_1, w_2, r) \in$ *current-level* **do**
for each $((w_1, w_2), X, S, (w_3, w_4)) \in F$ **do**
for each $c \in S$ **do**
if $(c \in R(w_1, w_i))$ **and** $(X = A)$
then *next-level* := *next-level* $\cup \{(w_3, w_4, r\&c)\}$;
$Q' := Q' \cup \{(w_3, w_4, r\&c)\}$;
$F' := F' \cup \{((w_1, w_2, r), X, c, (w_3, w_4, r\&c))\}$;
endif
if $(r = c\&t)$ **and** $(X = R)$
then *next-level* := *next-level* $\cup \{(w_3, w_4, t)\}$;
$Q' := Q' \cup \{(w_3, w_4, t)\}$;
$F' := F' \cup \{((w_1, w_2, c\&t), X, c, (w_3, w_4, t))\}$;
endif
endfor
endfor
endfor
endwhile
end

The state is characterized by the workstations of the head trailer and the tail trailer, respectively, and the sequence of segments that, in this state, the conveyor-load maintains allocated; $F' \subseteq Q' \times \{A, R\} \times SG \times Q'$ is the set of arcs that represents the transition from a state to another by the movement of

the head trailer or tail trailer. The movement of the head trailer allocates (A) the segment specified in the arc. Observe, that now in the RG_i a path from a state (w, w, ϵ), $w \neq w_i$, that corresponds to the birth of a conveyor-load in the workstation w, to the state (w_i, w_i, ϵ), represents the routing of a conveyor-load in the warehouse distribution center from the source workstation w to the destination workstation w_i. So, in order to obtain this $RG_i = (Q', F')$ from the corresponding $CBG_i(Q, F)$, we apply the algorithm 2 *(Note: with the symbol &, in the algorithm, we denote the concatenation operation of two strings)* In the

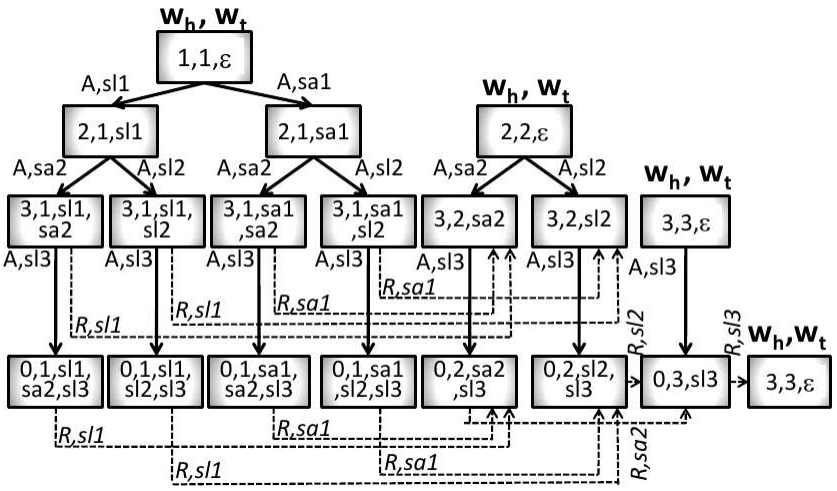


Fig. 5. Routing Graph for destination workstation 0.

Fig. 5 the RG_0, obtained from the CBG_0 of the Fig. 4. Applying the previous algorithm, we use the solid arcs to represent the segment allocation, and the dashed arcs to represent the segment release.

5 The Petri Net Model

In the previous section we have obtained the *RAS abstraction* of the warehouse distribution center plus the considered path selection algorithm. This abstraction is composed by the *resources*: The set SG of segments; and the set of *processes*: the set of routing processes to a destination workstation, each one represented by means of the corresponding Routing Graph. From these elements, in this section we proceed to the construction of a Petri Net integrating all processes and all resources.

First, from the $RG_i(Q', F')$, we construct the Petri Net $\mathcal{N}_i = \langle P_{0i} \cup Ps_i, T_i, F_i \rangle$ representing the state space of a conveyor-load born in the warehouse distribution center with destination workstation w_i. This construction proceeds according to the following rules.

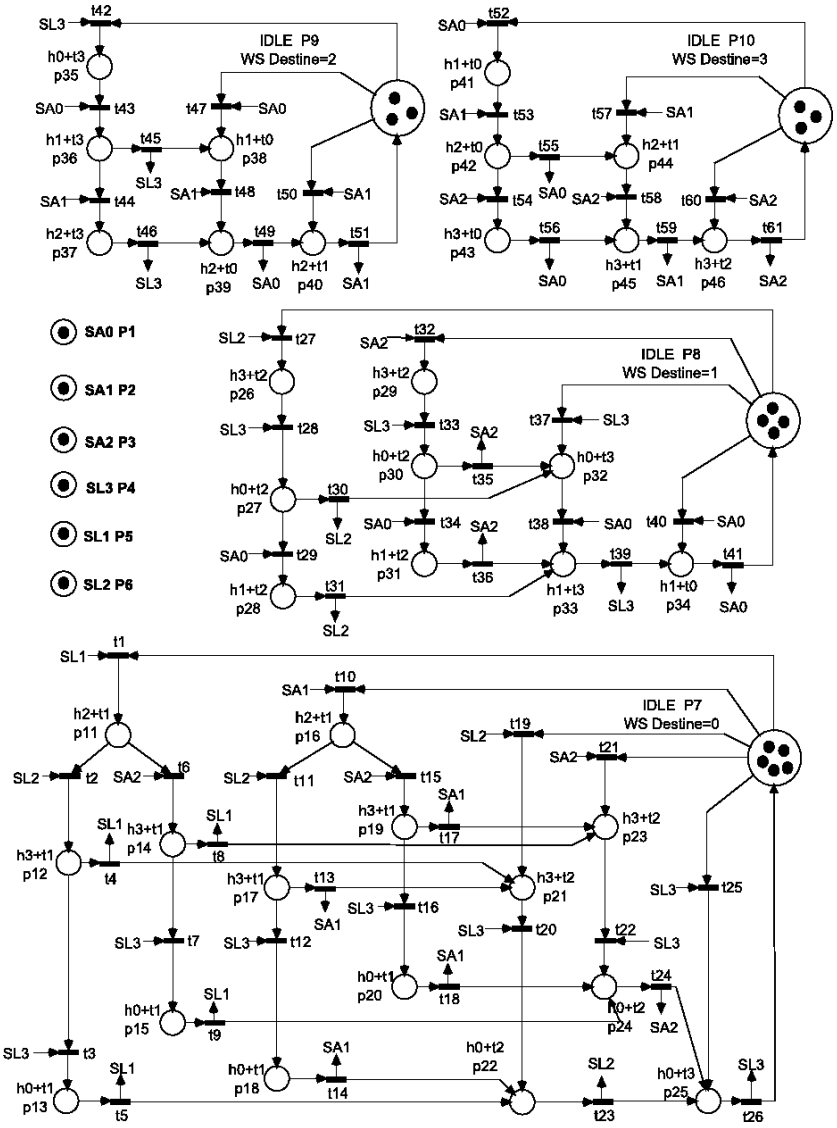


Fig. 6. Petri Net model for the example of the Fig. 1.

1. Add a place to the set Ps_i for each vertex of the RG_i, $(w_1, w_2, s) \in Q'$ such that $w_1 \neq w_2$. The name of the place will be formed by the concatenation of identifiers of the workstations of the head and tail trailer, w_1 and w_2, respectively, and the sequence of segments that remain allocated for this conveyor-load and represented by s. All these places are unmarked at the initial marking M_0, because in the initial state there are not conveyor-loads in transit. We call these places *process places*.

2. Add a unique place po_i, $Po_i = \{po_i\}$, corresponding to the fusion of states of the form $(w, w, \epsilon) \in Q'$. The initial marking of this place will be equal to the maximum number of conveyor-loads that can be simultaneously in transit to the destination workstation w_i. If this number is not limited, or it its unknown, then we don't need to add a place po_i, i.e. the number of conveyor-loads with destination workstation w_i, in this network, is only limited by the available segments. These places will be called *idle places.*
3. Add a transition to the set T_i for each arc of the graph RG_i. For an arc $((w_1, w_2, s), X, c, (w_3, w_4, r)) \in F'$, the name of the transition will be $w_1\&w_2\&s\&w_3\&w_4\&r$. (The concatenation of this strings identifying the elements of the arcs).
4. For each arc $((w_1, w_2, s), X, c, (w_3, w_4, r)) \in F'$, $w_1 \neq w_2$, add an arc from the place $w_1\&w_2\&s$ to the transition $w_1\&w_2\&s\ w_3\&w_4\&r$, and an arc from this transition to the place $w_3\&w_4\&r$.
5. For each arc $((w, w, s), X, c, (w_3, w_4, r)) \in F'$ add an arc from the *idle* place $p0_i$ (if there exists) to the transition $w\&w\&s\&w_3\&w_4\&r$, and an arc from this transition to the place w_3, w_4, r.

Observe that the net $\mathcal{N}_i$, obtained following the rules of the preceding paragraphs, is a strongly connected state machine. In effect, by construction, each transition has only one input place and only one output place because a transition has been added for each arc in the graph RG_i, and the places correspond to the both ends of the directed arc. Moreover, it is a strongly connected state machine because all vertex in RG_i, (w_1, w_2, s), is reachable by a path from a source vertex (w, w, ϵ), since the construction of RG_i requires that we can construct the sequence of allocated segments s from a source vertex; and from a vertex (w_1, w_2, s) always exists a path to the destination vertex (w, w, ϵ). Taking into account that place P_{0_i} represent the fusion of all vertices (w, w, ϵ) of the graph RG_i, we can conclude that the net $\mathcal{N}_i$ is strongly connected. Additionally, we can see that all circuits of $\mathcal{N}_i$ contain the place p_0, because the original RG_i is acyclic. After all these transformations we obtain a set of strongly connected state machines $\mathcal{N}_i$, each one corresponding to a different destination workstation in the warehouse distribution center. The last step to obtain the RAS view of the warehouse distribution center is the addition of the resources, that, in this case, they are the segments connecting the workstations, and their integration with to the state machines. This can be done state machine by state machine and constructing the full model by fusion of the resource places with the same name. That is, we are constructing the model in modular way. The two steps to be applied are:

1. Add a place p_c to the set P_R for each segment $c \in SG$ of in the warehouse distribution center. The initial marking of this place will be equal to the maximum number of trailers that can be in transit simultaneously in the segment. *(Normally. it will be equal to one representing tha availability of the segment).*
2. For each arc of the graph RG_i of the form $((w_1, w_2, s), A, c, (w_3, w_4, r)) \in F'$, add an arc from the place p_c, (resource place representing the segment

c) to the transition $w_1\&w_2\&s\&w_3\&w_4\&r$. This arc, in the Petri Net, will represent the allocation of the segment c. For each arc of the graph RG_i of the form $((w_1, w_2, s), R, c, (w_3, w_4, r)) \in F'$ add an arc from the transition $w_1\&w_2\&s\&w_3\&w_4\&r$ to the place c. This arc in the Petri Net represents the release of the segment c.

We denote this net by $\mathcal{N}_i^R$, representing the routing of the conveyor-loads to the destination workstation w_i, and the competition for the resource/segments. The full model is obtained from the different $\mathcal{N}_i^R$ by the fusion of the resource places (*segments places*) with the same name that appear in different $\mathcal{N}_i^R$. The Fig. 6 represents, in an schematic way, the full Petri Net corresponding to the example in Fig. 2. In this Fig. 2 the names of places and transitions have been simplified in order to maintain readable. In order to identify the states of the conveyor-loads with the places that represent them, we have used the simplified notation $h_i{+}t_j$; that it means that the head trailer is in workstation i and the tail trailer is in workstation j.

The final part of this section is devoted to prove that the obtained Petri Nets for warehouse distribution centers with minimal path selection algorithms belong to the subclass of Petri Nets named S^4PR [6][9]. In order to do that we recall the basic definitions of this class of nets.

Definition 1 (The class of S^4PR nets) *Let $I_{\mathcal{N}}$=$\{1, 2, ..., m\}$ be a finite set of indices. An S^4PR net is a connected generalised self–loop free Petri net $\mathcal{N}$=$\langle P, T, \mathbf{C}\rangle$ where:*

1. *$P = P_0 \cup P_S \cup P_R$ is a partition such that:*
   (a) *$P_S = \bigcup_{i\in I_{\mathcal{N}}} P_{S_i}$, $P_{S_i} \neq \emptyset$ and $P_{S_i} \cap P_{S_j} = \emptyset$, for all $i \neq j$.*
   (b) *$P_0 = \bigcup_{i\in I_{\mathcal{N}}} \{p_{0_i}\}$.*
   (c) *$P_R = \{r_1, r_2, \ldots, r_n\}$, $n > 0$.*
2. *$T = \bigcup_{i\in I_{\mathcal{N}}} T_i$, $T_i \neq \emptyset$, $T_i \cap T_j = \emptyset$, for all $i \neq j$*
3. *For all $i \in I_{\mathcal{N}}$, the subnet $\mathcal{N}_i$ generated by $P_{S_i} \cup \{p_{0_i}\} \cup T_i$ is a strongly connected state machine, such that every cycle contains p_{0_i}.*
4. *For each $r \in P_R$ there exists a minimal P–Semiflow, $\mathbf{y}_r \in \mathbb{N}^{|P|}$, such that $\{r\} = ||\mathbf{y}_r|| \cap P_R$, $\mathbf{y}_r[r] = 1$, $P_0 \cap ||\mathbf{y}_r|| = \emptyset$, and $P_S \cap ||\mathbf{y}_r|| \neq \emptyset$.*
5. *$P_S = \bigcup_{r\in P_R} (||\mathbf{y}_r|| \setminus \{r\})$.*

Each place p_{0_i} is called *idle place*. Places of P_R are called *resource places* being unique for the whole model. The Places of P_S are called *process places*. This definition must be completed with the definition of the *acceptable initial markings*. Initial markings represent no activity in the system, allowing the routing of each conveyor-load in isolation.

Definition 2 *Let $\mathcal{N} = \langle P_0 \cup P_S \cup P_R, T, \mathbf{C}\rangle$ be a S^4PR net. An initial marking $\mathbf{m_0}$ is acceptable for $\mathcal{N}$ if and only if: (1) $\forall i \in I_{\mathcal{N}}$, $\mathbf{m_0}[p_{0_i}] > 0$. (2) $\forall p \in P_S$, $\mathbf{m_0}[p] = 0$. (3) $\forall r \in P_R$, $\mathbf{m_0}[r] \geq \max_{p\in||\mathbf{y}_r||\setminus\{r\}} \mathbf{y}_r[p]$.*

From the previous definitions and the procedures described in the sections 4 and 5 to obtain the Petri Net model of an warehouse distribution center the following result can be easily verified.

Proposition 1 *Given an warehouse distribution center specified by means of a framework and a minimal adaptive routing algorithm, the Petri Net model obtained through the procedure described in sections 4 and 5, belongs to the class of S^4PR net systems.*

Proof (Sketch of the proof). In the section 5, after the rules to obtain the Petri Nets $\mathcal{N}_i$ from the corresponding Routing Graph RG_i, we have proven that each $\mathcal{N}_i$ is a strongly connected state machine, and for all $\mathcal{N}_i$,$\mathcal{N}_j$, $i \neq j$, they are disjoint net systems. We have also proven that every cycle of each strongly connected state machine $\mathcal{N}_i$ contains P_{0_i}. Therefore, to complete the proof we only need to prove the existence of a unique *p-semiflow* $\mathbf{y}_r \in \mathbb{N}^{|P|}$ for each resource r. But this is very easy to proof because from each transition where the resource place r inputs *(the resource is allocated)*, there exists a unique path, in the strongly connected state machine, to reach each transition where r outputs *(the resource is released)*. Moreover, all transitions where r is an output place in the state machine $\mathcal{N}_i$ are connected by means of a minimal path from some transition where r is an input place. Therefore, the resource r plus all the process places defining the minimal paths connecting the output transitions of r and the input transitions of r form the *p-semiflow* that it is unique because we are dealing with the nets $\mathcal{N}_i$ that they are state machines.

Observe that the previous result is also true for non-regular frameworks because we are considering in an explicit way the paths to a destination workstation. Therefore, non regularity does not affect the final Petri Net. Nevertheless, non-minimality of the path selection algorithms can lead to more general class of nets than the S^4PR in the case of existence of cycles in the followed route by some conveyor-load. Once we have characterized the type of nets we can obtain, we can use the developed theory for S^4PR, trying to interpret these results from the point of view of the warehouse distribution centers, in the next section. In some cases we will see that we arrive to some negative results.

6 The Analysis and synthesis phase

The Petri Net model obtained in the previous section belong to the S^4PR class. Therefore, we can take advantage of this property and use the theoretical results about the liveness characterization in S^4PR. One of this results is presented in the following theorem.

Theorem 2 ([6]) *An S^4PR, $\langle \mathcal{N}, \mathbf{m_0} \rangle$, is non–live if and only if there exists a marking $\mathbf{m} \in \mathrm{RS}(\mathcal{N}, \mathbf{m_0})$ such that the set of $\mathbf{m}$–process–enabled transitions is non–empty and each one of these transitions is $\mathbf{m}$–resource–disabled.*

This characterization is a state based characterization. The interpretation in terms of the warehouse distribution center is very easy. A token in a process place of the state machine $\mathcal{N}_i$ represent a conveyor-load in an intermediate workstation with destination workstation w_i. That is, is a conveyor-load in transit. The theorem 2 says that if all conveyor-loads in transit cannot advance because there is no an available segment to advance *(each one of these transitions is not enabled because an input resource place is empty)*, this situation characterizes a deadlock state: none of these conveyor-loads will arrive to its destination workstation because they are stopped forever in the current process places. In [6], verification procedures of the characterization stated in this theorem are presented. They are based in Integer Linear Programming Techniques.
An equivalent characterization to the previous one is based in the Petri Net concept of siphon. A siphon is a set of places that if they become a set of empty places, they remain empty forever *(these is a structural definition of siphon but we prefer to present the deep reason for the appearing of deadlocks in this class of nets)*. Therefore, all output transitions of the places of the empty siphon will be dead forever because at least an input place *(that belong to the siphon)* is empty forever. The presence of one of this siphons in the net is potentially bad because this siphon can become an empty siphon. The verification procedures search for a siphon and a reachable marking under which the siphon is empty. Empty siphons represent a generalization of the circular waits, because in a siphon we can find an intricate structure of superposed cycles of empty resources. For the Petri Net in Fig. 6, you can find the two following bad siphons D_i=$\{p_1, p_2, p_3, p_4, p_{13}, p_{15-to-20}, p_{22}, p_{24}, p_{25}, p_{28}, p_{30}, p_{31}, p_{33}, p_{34}, p_{36}, p_{37}, p_{39}, p_{40}, p_{42}, p_{43}, p_{45}, p_{46}\}$ and D_j=$\{p_1, p_2, p_3, p_4, p_6, p_{13}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}, p_{20}, p_{22}, p_{24}, p_{25}, p_{27}, p_{28}, p_{30}, p_{31}, p_{33}, p_{34}, p_{36}, p_{37}, p_{39}, p_{40}, p_{42}, p_{43}, p_{45}, p_{46}\}$. The deadlock state described in section 2 corresponds to the reachable marking written as a symbolic sum $m_r = p_5 + p_6 + 5 \cdot p_7 + 2 \cdot p_8 + \cdot p_9 + 2 \cdot p_{10} + p_{29} + p_{32} + p_{38} + p_{44}$. The reader can easily verify that the siphon D_i is insufficiently marked or he/she can verify the m_r satisfy the conditions of the theorem 2. Therefore, we conclude that the proposed path selection algorithm is not deadlock-free. After the previous analysis phase, the theory of S^4PR nets gives results and methods to enforce the liveness in the case of nets presenting deadlock states. These techniques transform the initial Petri Net model in such a way that deadlock states become not reachable. In some sense, they correspond to deadlock prevention techniques. We can incorporate this phase because we are using Petri Nets as formal model and they belong to the subclass of S^4PR. The known synthesis approaches enforcing liveness work on the bad siphons that can be found in the Petri Net model. These techniques can be classified into two groups.

1. **Centralized Approach:** [6][9] These techniques compute a place for each bad siphon preventing that the siphon becomes empty. This new place is of the same category that the resource places, and so it is said that the synthesis problem is solved by using virtual resources that they are implemented as a centralized monitors in the central software. In the case of the Petri Net of the Fig. 6 we need three places to make live the net. In fact, in some

cases, to take the decision to allocate the virtual resource/segment in a local workstation we can need coordinate the local path selection algorithm with other local routing algorithms.
2. **Distributed Approach:**[10]. Previous limitations are solved developing a distributed control policy using the so called *swap virtual segments*.

All these methods are iterative, but the performed transformations maintain the transformed Petri Net inside the class of S^4PR nets.

7 Conclusions

The design of deadlock-free minimal adaptive routing algorithms for warehouse distribution centers is a complex and tedious task, for which the current methodologies, in many cases, only supply trial and error procedures. The assistance to the designer is very small in order to fix the problem in the proposed algorithm. In this paper we propose a methodology oriented to the design of deadlock-free minimal adaptive routing algorithms trying to cope with all phases of the design. The first step in this methodology consists of the *abstraction* of the system in order to retain only the elements of the system allowing the study of the appearing of deadlocks. These elements are the segments of the warehouse distribution center, that they are seen as the resources for which the routing processes compete to send conveyor-loads to destination workstations. The other elements are the routing processes itself that represent the routing sequence through the framework according to the routing algorithm. The result of this abstraction process is formalized by means of a Routing Graph for each possible destination workstation. From the Routing Graphs and the segments we have obtained Petri Nets that, for the class of routing algorithms that we are considering, belong to the class of S^4PR. Therefore, we profit that the class of S^4PR is a well studied subclass of Petri Nets and using the known results we can proceed with the analysis and synthesis phases of our methodology. So, the deadlock-free property in the warehouse distribution center correspond to the liveness-property in our Petri Net model. The analysis of this liveness property can be done by two alternative characterizations that have a good interpretation at the level of warehouse distribution center. Algorithms and methods to verify the property can be found in [6]. In the case of non-liveness, there exist methods to enforce the liveness property based in the addition of places that can be interpreted in terms of Petri Net model as centralized software monitors.

References

1. Fanti, M.: A deadlock avoidance strategy for AGV systems modelled by coloured Petri nets. In: Discrete Event Systems, 2002. Proc. Sixth Int. Workshop on. (2002) 61 – 66
2. Wu, N., Zhou, M.: Resource-oriented Petri nets in deadlock avoidance of AGV systems. In: Robotics and Autom., 2001. Proc. 2001 ICRA. IEEE Int. Conf. on. Volume 1. (2001) 64 – 69 vol.1

3. Wu, N., Zhou, M.: Modeling and deadlock control of automated guided vehicle systems. Mechatronics, IEEE/ASME Transactions on **9**(1) (2004) 50 –57
4. Wu, N., Zhou, M.: Modeling and deadlock avoidance of automated manufacturing systems with multiple automated guided vehicles. Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on **35**(6) (2005) 1193 –1202
5. Wu, N., Zhou, M.: Shortest routing of bidirectional automated guided vehicles avoiding deadlock and blocking. Mechatronics, IEEE/ASME Transactions on **12**(1) (2007) 63 –72
6. Tricas, F.: Analysis, Prevention and Avoidance of Deadlocks in Sequential Resource Allocation Systems. PhD thesis, Zaragoza. España, Departamento de Ingeniería Eléctrica e Informática, Universidad de Zaragoza (2003)
7. Tricas, F., Colóm, J., Ezpeleta, J.: A Solution to the Problem of Deadlocks in Concurrent Systems Using Petri Nets and Integer Linear Programming. In Horton, G., Moller, D., Rude, U., eds.: Proc. of the 11th European Simulation Symp., Erlangen, Germany, The society for Computer Simulation Int. (1999) 542–546
8. Park, J., Reveliotis, S.A.: Enhancing the Flexibility of Algebraic Deadlock Avoidance Policies Through Petri Net Structural Analysis. In: Proc. of the 2000 IEEE Int. Conf. on Robotics and Autom., San, Francisco, USA (2000) 3371–3376
9. Tricas, F., García-Vallés, F., Colóm, J.M., Ezpeleta, J.: A Petri Net Structure-Based Deadlock Prevention Solution for Sequential Resource Allocation Systems. In: Proc. of the 2005 IEEE Int. Conf. on Robotics and Autom., Barcelona, Spain (2005) 272–278
10. Rovetto, C., Cano, E., Colóm, J.: Liveness Enforcing Methods for Resource Allocation Distributed Systems using Petri Nets, Research Report RR-056-09. Depto de Informática e Ingeniería de Sistemas., University of Zaragoza (2009)

Acknowledgement

This work has been partially supported by the European Community's Seventh Framework Programme under project DISC (Grant Agreement n. INFSO-ICT-224498). This work has been also partially supported by the project CICYT-FEDER DPI2006-15390.

Taming the Shrew – Resolving Structural Heterogeneities with Hierarchical CPNs*

M. Wimmer[1], G. Kappel[1], A. Kusel[2],
W. Retschitzegger[2], J. Schoenboeck[1], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

Abstract. Model transformations play a key role in the vision of Model-Driven Engineering (MDE) whereby the overcoming of structural heterogeneities, being a result of applying different meta-modeling constructs for the same semantic concept, is a challenging, recurring problem, urgently demanding for reuse of transformations. In this respect, an approach is required which (i) abstracts from the concrete execution language allowing to focus on the resolution of structural heterogeneities, (ii) keeps the impedance mismatch between specification and execution low enabling seamless debuggability, and (iii) provides formal underpinnings enabling model checking. Therefore, we propose to specify model transformations by applying a set of abstract mapping operators (MOPs), each resolving a certain kind of structural heterogeneity. For specifying the operational semantics of the MOPs, we propose to use Transformation Nets (TNs), a DSL on top of Colored Petri Nets (CPNs), since it allows (i) to keep the impedance mismatch between specification and execution low and (ii) to analyze model transformations by evaluating behavioral properties of CPNs.

Key words: Model Transformation Reuse, Hierarchical CPNs, Structural Heterogeneities, Mapping

1 Introduction

MDE is a current trend in software engineering where models are used as first-class artifacts throughout the software lifecycle [2], which are then systematically transformed to concrete implementations. In this respect, model transformations play a vital role, representing *the* key mechanism for *vertical transformations* like the generation of code and *horizontal transformations* like model exchange between different modeling tools, to mention just a few. In the context of transformations between different metamodels and their corresponding models, the overcoming of *structural heterogeneities*, being a result of applying

* This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13.

different meta-modeling constructs for the same semantic concept [11, 13] is a challenging, recurring problem, urgently demanding for reuse of transformations.

In this respect, reusable transformations should abstract from a concrete transformation language, allowing to (preferably graphically) specify transformations in an explicit *specification view* without having to struggle with the intricacies of a certain transformation language. Secondly, for being able to debug and comprehend resulting specifications, the impedance mismatch between the specification view and the executable formalism needs to be minimized, demanding for a *debugging view* which retains the structure of the specification view, i.e., components used in the specification view should not get scattered in the debugging view. Finally, since debugging can only provide limited evidence of correctness by means of a set of test runs, the underlying executable formalism for the *execution view* should provide means to enable *model checking* [3].

We therefore propose to specify horizontal model transformations by means of abstract mappings representing a set of reusable transformation components, called mapping operators (MOPs), to resolve recurring structural heterogeneities. These MOPs operate on different levels of granularity, i.e., we provide a set of *kernel MOPs* representing the basic functionality needed for resolving structural heterogeneities and a set of *composite MOPs* encapsulating several kernel MOPs, thus enhancing scalability of our approach. In order to specify the operational semantics of the MOPs, we propose to use TNs [23], a DSL on top of CPNs [9], since TNs allow to keep the impedance mismatch between specification view and debugging view low by encapsulating the transformation logic of a single MOP together with the metamodels and the models. Thereby debuggability and comprehensibility are fostered, i.e., the ability of finding and reducing the number of bugs. Moreover, the underlying CPNs allow to specify reusable components in the form of modules, which can be nested in a hierarchical way, allowing to accordingly represent composite MOPs. Therefore the main contribution of this paper is to enable reuse also on the execution level, i.e., the Petri Net layer. Finally, the formal underpinnings of CPNs allow the application of generally accepted behavioral properties to analyze the transformation specification. The whole framework is called TROPIC – TRansformations On Petri nets In Color.

The remainder of this paper is structured as follows. Section 2 introduces a motivating example, Section 3 concentrates on the specification of a transformation and Section 4 deals with the debugging thereof. The subsequent Section 5 shows how TNs are represented in standard CPNs and how behavioral properties are exploited to analyze the transformation specification. Lessons learned are discussed in Section 6 and related work is surveyed in Section 7. Finally, Section 8 concludes the paper with an outlook on future work.

2 Motivating Example

Structural heterogeneities between different metamodels occur due to the fact that semantically equivalent concepts can be expressed by different metamodeling concepts, e.g., explicitly by classes or only by attributes. Fig. 1 shows an

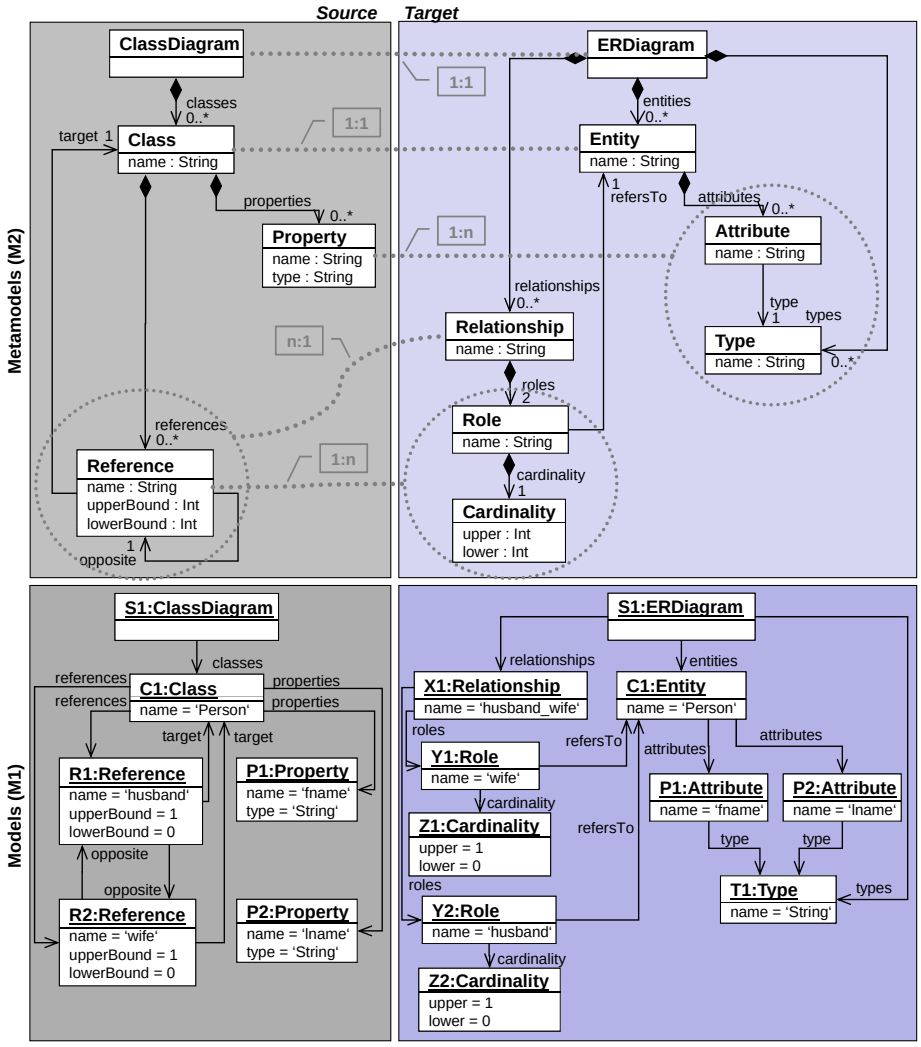


Fig. 1. Metamodels and Models of the Running Example

example used throughout the rest of the paper which exhibits common structural heterogeneities between metamodels, applying different modeling constructs to represent relationships as can be found e.g., in Ecore[3] or in Entity-Relationship Models. The `ClassDiagram` shown on the left side of Fig. 1, only provides unidirectional references, thus bidirectionality needs to be modeled by a pair of opposite references. In contrast to that, the `ERDiagram` explicitly represents bidirectionality, allowing to express relationships in more detail, e.g., using roles.

[3] http://www.eclipse.org/modeling/emf/

In the following, the main correspondences between the `ClassDiagram` and the `ERDiagram` are shortly described. On the level of classes, three main correspondences can be recognized, namely *1:1 correspondences*, *1:n correspondences* and *n:1 correspondences*, which are also indicated by dotted lines in Fig. 1. 1:1 correspondences can be found (i) between the root classes `ClassDiagram` and `ERDiagram` as well as (ii) between `Class` and `Entity`. Regarding 1:n correspondences, again two cases can be detected, namely (i) between the class `Property` and the classes `Attribute` and `Type` and (ii) between the class `Reference` and the classes `Role` and `Cardinality`. Although these are two occurrences of a 1:n correspondence, there is a slight difference between them, since in the first case only for distinct values of the attribute `Property.type`, an instance of the class `Type` should be generated. Finally, there is one occurrence of a n:1 correspondence, namely between the class `Reference` and the class `Relationship`. It is classified as n:1 correspondence, since for *every pair* of `References`, that are opposite to each other, a corresponding `Relationship` has to be established. Considering attributes, only 1:1 correspondences occur, e.g., between `Class.name` and `Entity.name`, whereas regarding references, 1:1 correspondences and 0:1 correspondences can be detected. Concerning the first category, one example thereof arises between `ClassDiagram.classes` and `ERDiagram.entities`. Regarding the latter category, e.g., the relationship `ERDiagram.types` exists in the target without any corresponding counterpart in the source.

3 Specification View

As mentioned before, the actual specification of a transformation problem should abstract from a concrete transformation language allowing the transformation designer to focus on the resolution of structural heterogeneities without having to struggle with the intricacies of a certain transformation language. Therefore we propose to specify model transformations by means of abstract mappings being a declarative description of the transformation, as known from the area of data engineering [1]. For this we provide a library of composite MOPs [21]. Thereby we identified typical mapping situations being 1:1 copying, 1:n partitioning, n:1 merging, and 0:1 generating of objects, for which different MOPs are provided. In this respect, reuse is leveraged as the proposed MOPs are generic in the sense that they abstract from concrete metamodel types since they are typed by the core concepts of current meta-modeling languages like Ecore or MOF (i.e., class, attributes, references). To further structure the mapping process we propose to specify mappings in two steps.

In a first step, *composite MOPs*, describing mappings between classes are applied, providing an abstract *blackbox-view* (cf. Fig. 2). Every composite MOP consists of so-called *kernel MOPs*, thus the composite behavior is realized by a set of basic building blocks. These kernel MOPs are responsible for resolving structural heterogeneities and therefore they have to be able to map classes, attributes, and references in all possible combinations and mapping cardinalities. In this respect, MOPs are provided for copying exactly one object, value, or link

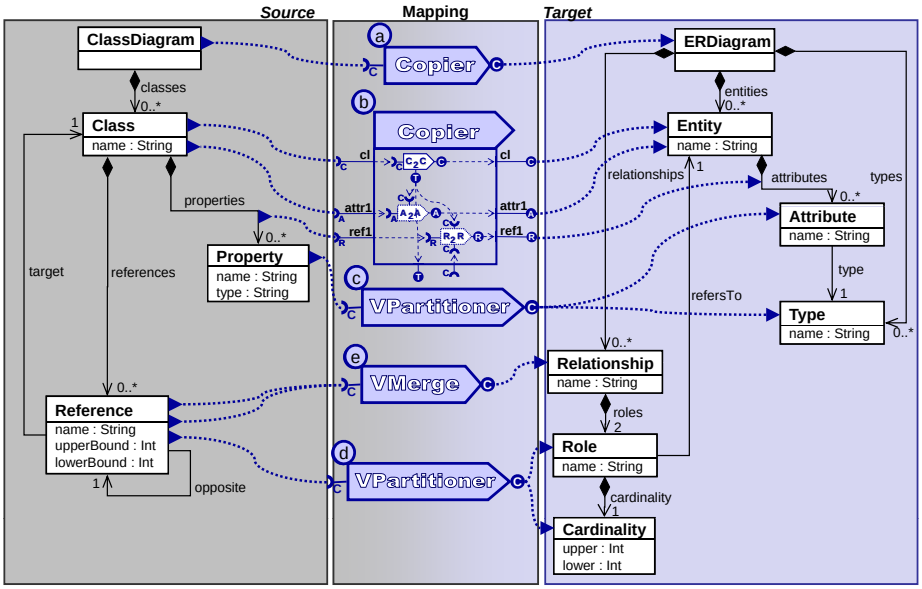


Fig. 2. Solution of the Running Example

from source to target, respectively (denoted as C(lass)$_2$C(lass), A(ttribute)$_2$A(ttribute), and R(eference)$_2$R(eference)). Moreover, MOPs are needed for merging objects, values, and links (denoted as $C_2^n C$, $A_2^n A$, and $R_2^n R$) resolving the structural heterogeneity that concepts in the source metamodel are more fine-grained than in the target metamodel. Finally, MOPs are needed for generating a target element without an obvious source element (denoted as 0_2C, 0_2A, and 0_2R) to resolve heterogeneities resulting from expressing the same modeling concept with different meta-modeling concepts – a situation which often occurs in metamodeling practice.[4] In a second step, the composite MOPs, which solely describe a mapping between classes at first, have to be refined to also map attributes and references in the so-called *whitebox-view* by the usage of kernel MOPs (cf. expanded Copier (b) in Fig. 2). Furthermore, kernel MOPs can be used to assemble new, user-defined composite MOPs.

As a concrete syntax for MOPs we are using a subset of the UML 2 component diagram concepts enabling the specification of model transformations in a plug & play manner. With this formalism, every MOP is defined as a dedicated component, representing a modular part of the transformation specification which encapsulates an arbitrary complex structure and behavior, providing well-defined interfaces. Every MOP has input ports with required interfaces (left side of the component) as well as output ports with provided interfaces (right

[4] Please note, that although composite MOPs for 1:n partitioning are provided, no additional kernel MOps are needed, since such situations can be simulated by n x 1:1 MOps.

Table 1. Overview of Composite MOPs used in the Example

Correspondence	MOP	Description	Composition of Kernel MOPs (EBNF)
1:1 - copying	Copier	creates exactly one target object per source object	Copier: C_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }
1:n - partitioning	VerticalPartitioner	splits one source object into several target objects	VerticalPartitioner: Copier { ObjectGenerator | Copier }
n:1 - merging	VerticalMerger	merges several source objects to one target object	VerticalMerger: $C^n{}_2C$ { A_2A | $A^n{}_2A$ | 0_2A |R_2R | $R^n{}_2R$ | 0_2R }
0:1 - generating	ObjectGenerator	generates a new target object without corresponding source object	ObjectGenerator: 0_2C { A_2A | $A^n{}_2A$ | 0_2A | R_2R | $R^n{}_2R$ | 0_2R }

side of the component), typed to classes (C), attributes (A), and relationships (R) (cf. `Copier` (b) in Fig. 2). Since there are dependencies between MOPs, e.g., a value can only be set after the owning object has been created, MOPs dealing with the transformations of classes additionally offer a trace port (T) at the bottom providing *context information*, indicating which target object has been produced from which source object(s). This port can be used by dependent MOPs to access context information via required context ports (T). In case of MOPs dealing with the mapping of attributes the corresponding interface is shown via one port on top, or in case of MOPs dealing with the mapping of references via two ports, whereby the top port depicts the required source context and the bottom port the required target context (cf. `Copier` (b) in Fig. 2).

For solving the running example, several composite MOPs have been applied as can be seen in Fig. 2. Table 1 presents an overview of the used composite MOPs to solve the example as well as their composition of kernel MOPs. For a detailed classification and description of all available kernel as well as composite MOPs we refer to [21]. To resolve the 1:1 correspondences between `ClassDiagram` and `ERDiagram` as well as between `Class` and `Entity` in our example we applied two `Copiers` since for every source object a corresponding target object should be generated (cf. MOPs (a) and (b) in Fig. 2)). The whitebox-view of the `Copier` (b) thereby shows the mapping of class `Class` to class `Entity` using a C_2C MOP. Moreover, the attribute `Class.name` is mapped to the attribute `Entity.name` by using an A_2A MOP. Finally, the reference `Class.properties` is mapped to the reference `Entity.attribute` using a R_2R MOP. To split the attributes of the class `Reference` to the target classes `Role` and `Cardinality` a `VerticalPartitioner` is applied (cf. MOP (d) in Fig. 2). Besides this default behavior, aggregation functionality is sometimes needed as is the case when splitting the `Property` concept into the `Attribute` and `Type` concepts, since a `Type` should only be instantiated for distinct `Property.type` values (cf. MOP (c) in Fig. 2). To merge two `Reference` objects to a single `Relationship` object a `VerticalMerger` is applied (cf. MOP (e) in Fig. 2).

4 Debugging View

In the previous section we showed how structural heterogeneities can be resolved by applying MOPs resulting in a declarative mapping specification. In order

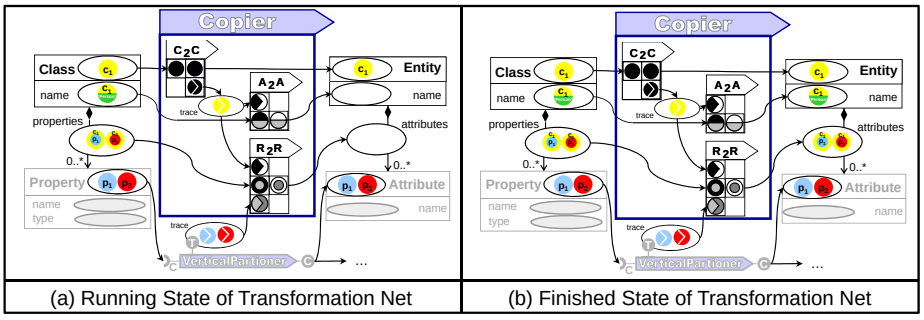
to execute this specification it has to be translated into an executable formalism, i.e., every MOP has to be assigned an operational semantics. Thereby, the impedance mismatch between the declarative specification and the actual operational semantics should be minimized in order to foster comprehensibility and debuggability. Since current transformation languages (cf. [4] for an overview) provide only a limited view on a model transformation problem, i.e., they do not visualize the actual metamodel and model being transformed, we proposed the TN formalism [23], being a DSL on top of CPNs [9]. The basic idea of TNs is to represent the transformation logic together with the metamodels and the models, whereby metamodel elements are represented by places, model elements by the according markings and the actual transformation logic by a system of transitions. Thus, an explicit runtime model is provided which can be used to observe the runtime behavior of a certain transformation. In the following we describe the core concepts of TNs as well as the adaptations introduced in comparison to standard CPNs to better suit the domain of model transformations.

Representation of Metamodels and Models. Since we rely on the core concepts of an object-oriented meta-metamodel the graph which represents the metamodel consists of classes, attributes, and references which are represented by according places in TNs. Therefore Fig. 3 depicts a place for the class `Class` as well as one place for the attribute `Class.name` and one place for the reference `Class.properties`. The graph which represents a conforming model consists of objects, data values and links which are represented by tokens in the according places. For every object that occurs in a model a one-colored *ObjectToken* is produced, which is put into a place that corresponds to the respective class in the source metamodel, e.g., the token `C1` in the `Class` place and the tokens `P1` and `P2` in the place `Property`, representing the objects of the source model depicted at the bottom of Fig. 1. The color is realized through a unique value that is derived from the object id (OID). For every value, two-colored *AttributeTokens* are produced whereby the upper color represents the object and the lower color the actual value, e.g., the `C1|Person` token represents the value "Person" of the attribute `Class.name` for the object `C1` in Fig. 3. Finally, for every link a two-colored *ReferenceToken* is produced. The outer color refers to the color of the token that corresponds to the owning object. The inner color is given by the color of the token that corresponds to the referenced target object, which is depicted by the corresponding tokens in the `Class.properties` place in Fig. 3.

Specification of Transformation Logic. The actual transformation logic is specified by means of a system of transitions and additional places, so-called *trace places* storing *context information* which reside in-between those places representing the original input and output metamodels. Transitions consist of so-called *query tokens* (LHS of the transition) representing the pre-condition of a certain transition, whereas *production tokens* (RHS of the transition) depict its postcondition. Thereby different query and production tokens for objects, values, links and context information are provided whose colors represent variables that are bound during execution, i.e, colors of query tokens are not the required colors for input tokens, instead they describe configurations that have to be fulfilled by

input tokens. In the copying sceanrio the color of the production tokens depend on the color of the query tokens, e.g., the production token and the query token of the C_2C transition exhibit the same color and therefore the source and the target object tokens exhibit the same color (cf. Fig. 3). However, it is also possible to produce a token of a not yet existing color if a target object is needed which does not directly correspond to a source object, e.g., in case a C_2^nC MOP which merges several source objects to a single new target object. Furthermore, to represent trace ports of MOPs, *trace places* containing *context tokens* indicate which target object has been created from which source object(s). Thereby the color(s) of the slot (left side) indicate(s) the used source object(s) whereas the generated target object is represented by the color of the remaining slice (right side of token). Since object tokens are simply copied in case of the depicted C_2C transition source and target context tokens exhibit equal colors (cf. Fig. 3(a)). Only if context information is available in a trace place, dependent transitions, e.g., the A_2A and R_2R transitions, are able to fire. For this, they query the context tokens in order to add a value or a link to the target object acquired from the context token (cf. Fig. 3(b)). Please note that in case of creating a new target object, source and target color of the context tokens differ from each other. Thus, dependent transitions must be able to cope with differently colored context tokens and therefore the context query tokens of the dependent A_2A and R_2R transitions in Fig. 3 show different colors (which are only variables and are therefore also able to match for same colored tokens).

Adaptations of Standard CPNs. In contrast to standard CPNs, TNs exhibit a different default firing behavior, i.e., tokens are not consumed per default (therefore source tokens are preserved in their corresponding source places). This is since all possible token combinations must be taken into account. For example, if the R_2R transition would consume `Class` tokens and `Property` tokens from the trace places (cf. Fig. 3), the transition could fire only once although multiple `Properties` would be available, since there is a 1:n relationship between `Class` and `Property`. Moreover, if more than one transition accesses a certain place, consuming firing behavior would lead to erroneous race conditions.



(a) Running State of Transformation Net (b) Finished State of Transformation Net

Fig. 3. Debugging View of Copier MOP

Summarizing, TNs provide a formalism to specify the operational semantics of the provided MOPs. Thereby TNs reduce the impedance mismatch between the abstract declarative mapping specification and the actual operational semantics since there is a 1:1 correspondence between kernel MOPs and transitions. Additionally, all artifacts in a model transformation, i.e., metamodel, transformation logic and the involved models are represented in a homogenous view. Furthermore, as query and production tokens are only typed to the core concepts of object-oriented metamodels (class, attributes and references) the specified transformation logic can be reused between arbitrary metamodels (as intended by the MOPs). Due to the fact that every MOP is realized by an independent set of transitions every MOP can be debugged individually, thus enabling a component-oriented debugging approach.

5 Execution View

Since TNs represent a DSL on top of CPNs they can be fully translated into existing CPN concepts to make use of efficient execution engines and their properties to analyze model transformations [20]. The actual translation is transparent to the user since a TN is automatically converted to an according CPN using the ASAP platform [19]. The ASAP platform provides an EMF-based implementation of the PNML standard[5] for CPNs. The CPN model can then be used to check the syntax of the corresponding TN, to simulate the TN and to calculate behavioral properties for the specified model transformations. Since every MOP is realized by an independent set of TN transitions we provide pre-defined hierarchical CPNs for kernel and composite MOPs, detailed in the following. Furthermore, the application of behavioral properties for analyzing model transformations is shown.

5.1 Representation of Kernel MOPs

Kernel MOPs and their respective operational semantics in TNs can be represented by means of *modules* or so-called *substitution transitions* in hierarchical CPNs whereby the *ports* of the substitution transitions are only typed by classes, attributes, and references. The ports are then bound to the corresponding *socket places* being the places derived from the source and target metamodel. In the following we show how to realize the non-consuming behavior in CPNs as well as the translation of kernel MOPs to hierarchical CPNs.

Adaptations of Standard CPNs. To realize the non-consuming firing behavior, a so-called *history place* is introduced for every transition. It stores all token combinations that have already been fired by this transition in a sorted list in order not to blow up the state space, i.e., there is no difference if token `P1` or token `P2` has been transformed first in our scenario. The history place is connected to the corresponding transition whereby a guard condition prevents the

[5] http://www.pnml.org/

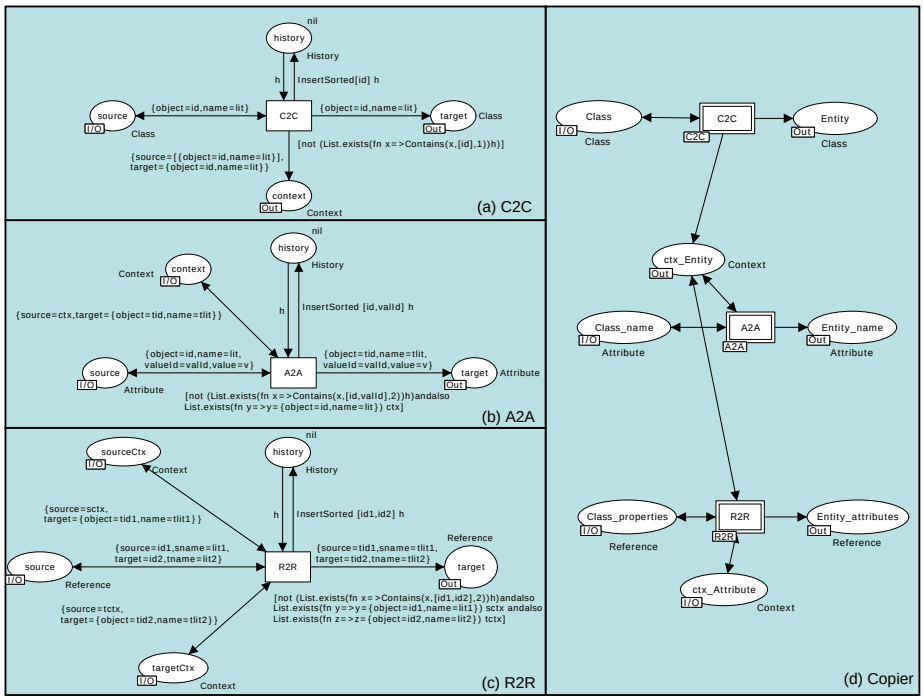


Fig. 4. Realization of MOPs by Hierarchical CPNs

transition from firing a certain token combination twice. Moreover, the standard arcs are replaced by so-called *test arcs*, which do not consume tokens from the connected input places. For further details on the translation of TNs to CPNs we refer the interested reader to [23].

MOPs mapping Classes. In case of kernel MOPs dealing with the mapping of classes, e.g., a C_2C MOP as depicted in Fig. 4(a), the in- and outports have to be typed to the colorset `Class` (`colset Class = record object : INT * name : STRING`). As a C_2C MOP simply copy tokens, the same arc inscription can be found on the in- and outgoing arcs (represented by the same colors of query and production token in TNs). Furthermore, kernel MOPs mapping classes provide context information stored in the *context* port[6]. The colorset `Context` thereby defines a record consisting of a list of classes (since more than one class can be used to enable the transition in case of a C_2^nC) and a target class (`colset Context = record source:SourceContext * target : Class; colset SourceContext = list Class;`).

[6] Note that ports providing context information in MOPs and CPNs are used to enable dependent MOPs or transitions, i.e, they provide required tokens to enable a transition, whereas the history concepts solely hinders multiple firings of transition in CPNs

MOPs mapping Attributes or References. In case of kernel MOPs dealing with the mapping of attributes or references, e.g., an A_2A MOP or R_2R MOP as depicted in Fig. 4(b) and (c), the ports have to be typed to the colorset `Attribute` and `Reference` respectively (`colset Attribute = record object:INT * name:STRING * valueId : INT * value : STRING; colset Reference = record source:INT * sname: STRING * target:INT * tname: STRING;`). Since attributes and references should only be transformed if the owning object of an attribute or the source and target objects of a reference have already been transformed, the guard condition of the transition not only prevents the multiple firing but additionally checks if the context place already contains the necessary context information. If the condition is fulfilled, an attribute or reference token is produced whereby the new owning object `tid` is acquired from the context tokens (cf. arc inscription at the in- and outgoing arcs from context places Fig. 4(b) and (c)). These hierarchical CPNs can then be assembled to more coarse-grained hierarchical CPNs to represent, e.g., a `Copier` as shown in Fig. 4(d). In the following, composite MOPs are elaborated in more detail.

5.2 Representation of Composite MOPs

Specification View. In Section 3 we introduced coarse-grained composite MOPs which encapsulate several kernel MOPs, e.g., a `Copier` consists of exactly one C_2C, and several MOPS for mapping attributes or references. As can be seen in Table 1, composite MOPs can not only consist of kernel MOPs but might encompass composite ones themselves, e.g., `VerticalPartitioner` which consists of a `Copier` and an `ObjectGenerator` (cf. Fig. 5(a)). In our running example this MOP was used to split the source concept `Property` into the concepts `Attribute` (achieved by the contained `Copier`) and `Type`, whereby a `Type` should only be instantiated for distinct `Property.type` values overcoming the heterogeneity that a concept is expressed as an attribute in the source metamodel and as a class in the target metamodel (achieved by the contained `ObjectGenerator`).

Debugging View. The relation between composite and the kernel MOPs can be seen in the debugging view (cf. Fig. 5(b)). First, the C_2C transition of the copier streams the corresponding object tokens, thus creating an `Attribute` for every `Property`. The thereby generated context information enables the A_2A transition in order to set the `Attribute.name` values. Second, the A_2C transition generates a `Type` object token for distinct `Property.type` values, which is indicated by the `distinctInputValue` annotation on the transition meaning that only context information in the according trace place is generated but no new target token in case that a value occurs several times. Therefore, the trace place of the `ObjectGenerator` composite MOP contains two `Property.type` tokens which both have been mapped to the same `Type` object (depicted by the equal target color of the context tokens) since both source tokens have the same value ‘‘`String`’’. In order not to produce too many attribute tokens the dependent A_2A MOP has to match only for distinct target colors of context tokens resulting in distinct output values (indicated by the according annotation in (cf.

Fig. 5(b)). Finally, the generated `Attribute` and `Type` objects have to be accordingly linked by the reference `Attribute.type`. Since there is no according source reference available we have to generate this reference by applying a 0_2R MOP. Nevertheless, the transformation designer has to define during specification how the generated target objects are related to each other in the source model. In our example the intention is to generate a reference for every `Attribute` object having set an according `Attribute.type` value. In order to get this input the `InputGen` transition collects the tokens and thereby generates (self) references. These references can then be processed by the `Linker` component which finally produces the according `Attribute.type` references.

Execution View. In order to represent the different levels of granularity, the corresponding hierarchical CPN again consists of several nested ones, thus leading to multi-level hierarchical CPNs. As shown in Fig. 5(c) the `VerticalPartitioner` consists of two substitution transitions, being a `Copier` and an `ObjectGenerator`. As already shown in the debugging view (cf. Fig. 5(b)), the main part of an `ObjectGenerator` is an A_2C kernel MOP. Since only for distinct values a new target object should be generated, an additional `values` place containing a list of records (`colset A2CList = list A2C; colset A2C = record value:INT`

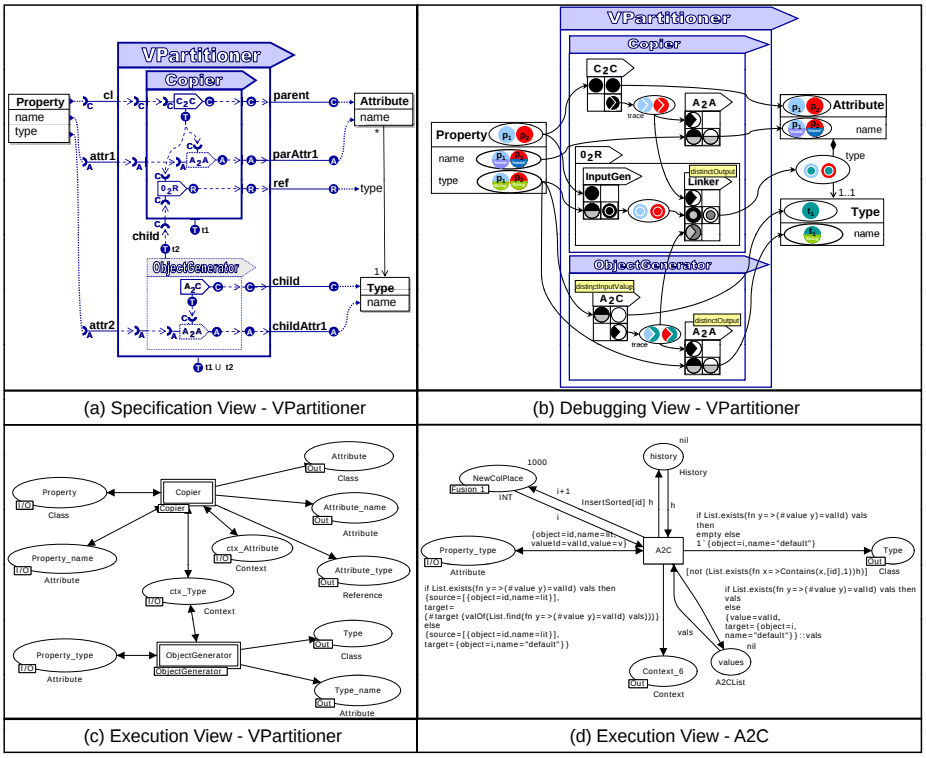


Fig. 5. Different Views of VerticalPartitioner

`* target:Class`), expressing which values have already been converted to a certain class token, is introduced (cf. Fig. 5(d)). The conditions on the outgoing arcs to the target place and to the `values` places ensure that a token is only created if the value has not been contained in the `values` list before. In contrast to that, context information is produced for any firing of the transition whereby the source object is connected to an already existing class target token if the values list contains an according entry, i.e., if a `Type` has already been created for a certain `Property.type` value. To represent the fact that a `Type` object has no according counterpart in the source model, we generate a new object id which is the task of the (fusion) place `NewColPlace` and the according arc inscriptions represented by a newly colored object production token in TNs.

5.3 Behavioral Properties to Analyze Mappings

Although the operational semantics of MOPs is predefined, configuration errors might occur when applying the MOPs in the specification phase leading to an erroneous interplay between MOPs. In the following we show how typical errors can be detected by means of behavioral properties of the underlying CPNs [20].

Model Comparison using Boundedness Properties. Typically, the first step in analyzing the correctness of a transformation is to compare the generated target model to an expected target model. To identify wrong or missing target elements in terms of tokens automatically, *Boundedness* properties can be applied. An example thereof could be the A_2C MOP in the above example which creates target tokens for distinct values only. Therefore dependent transitions need to generate a distinct output as well, e.g., to set the `Type.name` value only once. If this is not specified by the user accordingly, too many `Type.name` tokens are generated which can be detected by comparing the Boundedness properties of the according place of the generated model to the expected target model.

Checking Interplay of MOPs using Liveness Properties. Another source of error during the refinement of composite MOPs by kernel MOPs is the mapping of dependent attributes and references. In case that MOPs dealing with attributes and references are connected to wrong source or target context ports the corresponding transition is not able to fire which can be detected by *Liveness Properties* such as *Dead Transition Instances* or *L0-Liveness*.

Termination and Confluence Analysis using Dead and Home Markings. A transformation specification must always terminate, thus the state space has to contain at least one *Dead Marking*, which is typically ensured by the history concept. Moreover, it has to be ensured that a dead marking is always reachable, meaning that a transformation specification is confluent, which can be checked by the *Home Marking*. Furthermore, it is possible to check if a certain marking, i.e., the target marking derived from the expected target model, is reachable. If this marking is equal to the Dead and Home Marking it is ensured that the specified mapping always generates the expected target model.

6 Lessons Learned

This section presents lessons learned and discusses key features of our approach.

Kernel MOPs Enable Extensibility. Kernel MOPs form the basis for overcoming structural heterogeneities and thereby have to exhibit a well-defined operational semantics. Since composite MOPs are solely based on kernel MOPs, the composite operational semantics results from the operational semantics of the kernel MOPs. Therefore, the library of composite MOPs can be easily extended on basis of the kernel MOPs without the need of adapting the compilation to TNs and CPNs, respectively.

CPNs Allow for Parallel Execution. As CPNs exhibit an inherent concurrency, parallel execution of transformation logic is possible thereby increasing the efficiency of a transformation execution. In particular, mappings between classes are independent from each other and therefore the transformation of objects can be fully parallelized. The same is true for depending attributes and references which can also be transformed in parallel after the owning objects have been created and thus the needed context tokens are available.

Visual Formalism Eases Debugging and Understandability. TNs provide a visual formalism for defining model transformations which is especially useful for debugging purposes, since the actual execution of a certain model transformation can be simulated. In this respect, the transformation of model elements can be directly followed by observing the flow of tokens and therefore undesired results can be detected easily.

History Ensures Termination. As mentioned above, TNs introduce a specific firing behavior in that transitions do not consume the source tokens satisfying the precondition but hold them in a history. Thus, a transition can only fire once for a specific combination of input tokens prohibiting infinite loops, even for test arcs or cycles in the net. Only if a transition occurs in a cycle and if it produces new objects every time it fires, the history concept can not ensure termination. Such cycles, however, can be detected at design time and are automatically prevented for TNs. In contrast to model transformation languages based on graph grammars, where termination is undecidable in general [14], TNs ensure termination already at design time.

State Space Explosion Limits Model Size. A known problem of model checking and thus also of behavioral properties of Petri Nets is that the state space might become very large. Currently, the full occurrence graph is constructed to calculate properties leading to memory and performance problems for large source models and transformation specifications. Often a marking M has n concurrently enabled, different binding elements leading all to the same marking. Nevertheless, the enabled markings can be sorted in $n!$ ways, resulting in an explosion of the state space. As model transformations typically do not care about the order how certain elements are bound, the number of bindings can be reduced to 2^n bindings, thus enhancing scalability of our approach.

7 Related Work

In the following, related work is summarized according to the proposed views.

Specification View. In the area of model engineering only the ATLAS Model Weaver (AMW) [7] provides a dedicated mapping tool allowing the definition of model transformations independent of a concrete transformation language. By extending the weaving metamodel, one can define the abstract syntax of new weaving operators which roughly correspond to our MOPs. The semantics of weaving operators is determined by a higher-order transformation [16], taking a model transformation as input and generating another model transformation as output. Compared to our approach, the weaving models are compiled into low-level ATL [10] transformation code which is in fact a mixture of declarative and imperative language constructs. Thus, this solution exhibits an impedance mismatch, hindering the understanding and debugging of the resulting code.

Debugging View. Concerning model transformations in general, there is little debugging support available. Most often only low-level information available through the execution engine is provided, but traceability according to the higher-level correspondence specifications is missing. For example, in the Fujaba environment, a plugin called MoTE [18] compiles TGG rules [12] into Fujaba story diagrams that are implemented in Java, which obstructs a direct debugging on the level of TGG rules. In [8], the generated source code is annotated accordingly to allow the visualization of debugging information in the generated story diagrams, but not on the TGG level. Concerning the understandability of model transformations in terms of a visual representation and a possibility for a graphical simulation, only graph transformation approaches like Fujaba allow for a similar functionality. However, these approaches neither provide an integrated view on all transformation artifacts nor do they provide an integrated view on the whole transformation process in terms of the past state, i.e., which rules fired already, the current state, and the prospective future state, i.e., which rules are now enabled to fire. Therefore, these approaches provide snapshots of the current transformation state, only.

Execution View. Current transformation languages provide only limited support to analyze transformation specifications as summarized in the following. In the area of graph transformations some work has been conducted that uses Petri Nets to check properties of graph production rules. Thereby, the approach proposed in [17] translates individual graph rules into a Place/Transition Net and checks for its termination. Another approach is described in [6], where the operational semantics of a visual language in the domain of production systems is described with graph transformations. The production system models as well as the graph transformations are transformed into Petri Nets in order to make use of analysis techniques for checking properties of the production system models. Finally, a recent work by de Lara and Guerra [5] proposes to translate QVT-Relations into CPNs - on the one hand to provide a formal semantics for QVT Relations and on the other hand to analyze QVT Relations specifications - pursuing similar ideas as followed in our previous work [22]. Nevertheless, these approaches are using Petri Nets only as a back-end for analyzing properties of

transformations, whereas we are using a DSL on top of CPNs as a front-end for model transformations, thereby fostering debuggability.

8 Future Work

Currently, only the most important concepts of modeling languages, i.e., classes, attributes and relationships have been considered by our MOPs. It would be desirable, however, to extend our MOP library to be able to deal also with concepts such as inheritance or complex mathematical operations. Furthermore, only a basic prototype of the proposed debugging view is available. We therefore focus on improving our prototype, e.g., by accordingly visualizing the findings of the formal properties. Concerning verification support, we focused on small mapping scenarios up to now only, not least due to the state space explosion problem. Nevertheless the ASAP platforms provides the possibility to specify own algorithms to explore the state space which could additionally be adopted to the domain of model transformation to enable verification support for larger scenarios. To further support the transformation designer in complementing the mapping in the whitebox-view, auto-completion strategies should be incorporated. In this respect, we will investigate on matching strategies [15] which may be applied to automatically derive attribute and relationship mappings.

References

1. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proc. of SIGMOD'07*, pages 1–12. ACM, 2007.
2. J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2):31, 2005.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
4. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
5. J. de Lara and E. Guerra. Formal Support for QVT-Relations with Coloured Petri Nets. In *Proc. of MoDELS'09*, 2009.
6. J. de Lara and H. Vangheluwe. Automating the Transformation-Based Analysis of Visual Languages. *Formal Aspects of Computing*, 21, Mai 2009.
7. M. Del Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *SoSyM*, 8(3):305–324, 2009.
8. L. Geiger. Model Level Debugging with Fujaba. In *Proceedings of 6th International Fujaba Days*, pages 23–28, Dresden, Germany, 2008.
9. K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.
10. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.
11. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: A context-based approach. *The VLDB Journal*, 5(4):276–304, 1996.
12. A. Koenigs. Model Transformation with TGGs. In *Proc. of Model Transformations in Practice Workshop of MoDELS'05*, Montego Bay, Jamaica, 2005.

13. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In *Proc. of BTW'07*, 2007.
14. D. Plump. Termination of graph rewriting is undecidable. *Fundamental Informatics*, 33(2), 1998.
15. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.
16. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA'09*, pages 18–33, 2009.
17. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformation by Petri Nets. In *Proc. of ICGT'06*, pages 260–274, 2006.
18. R. Wagner. Developing Model Transformations with Fujaba. In *Proc. of the 4th Int. Fujaba Days 2006*, pages 79–82, 2006.
19. M. Westergaard and L. M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Proc. of the 30th Int. Conf. on Applications and Theory of Petri Nets*, pages 313–322, 2009.
20. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. Right or Wrong? - Verification of Model Transformations using Colored Petri Nets. In *Proc. of 9th DSM Workshop*, 2009.
21. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. of ICMT'10*, 2010.
22. M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger. Reviving QVT Relations: Model-based Debugging using Colored Petri Nets. In *Proc. of MoDELS'09*, pages 727–732, 2009.
23. M. Wimmer, A. Kusel, J. Schönböck, T. Reiter, W. Retschitzegger, and W. Schwinger. Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Proc. of PNSE'09*, pages 35–50, 2009.

Part IV

Poster Abstracts

MATLAB / Simulink and Program Sketcher for Verification of Hybrid Petri Nets Implementation into Programmable Logic Controller

Luděk Chomát, Petr Malounek, Petr Pivoňka

Brno University of Technology, Faculty of Elektrotechnical Engineering and Communication, Department of Control and Instrumentation, 2906/4 Kolejní, 612 00 Brno, Czech Republic
xchoma00@phd.feec.vutbr.cz

Abstract: The main goal of this work-in-progress is to find a suitable tool for modeling and verification of Hybrid Petri Nets. A suitable mathematical tool is MATLAB/Simulink for which is created PN_CLGtoolbox in CLG laboratory. The created tool together with already existing program Sketcher will be used to explore and consider different approaches for engine control modeling.

Keywords: MATLAB / Simulink, PLC, Sketcher, Petri Net, Motor, Implementation and Verification.

1 Introduction

Currently, we have high demands on engine control and synchronization between them. Petri nets are a suitable tool for parallel system modeling which can be used for engine control. Essentially, it is possible to control any power (DC, synchronous, asynchronous, stepper, etc.) with approximately the same control algorithm. However, it is necessary to find a similar basic structure of control that can be implemented by hybrid Petri nets. Programmable logic controllers in conjunction with Petri nets are appropriate tools for managing dynamic systems. At the beginning, it is important to identify and verify hybrid Petri net [1], which will help us in finding the basic structure for engines.

2 Identification and Verification of Petri Nets

Figure 1. shows the Petri nets process scheme and deployment to a real technological process. Here we see that at first, it is necessary to propose a hybrid Petri net, which we simulate to determine the proper functioning of the net, prior to implementation of the real process. Using simulations, we identify characteristics of hybrid Petri nets, which we compare with the identification of hybrid Petri nets from the programmable controller or other platforms. For identification we use MATLAB /

Simulink. Verification of hybrid Petri nets is very important for safe working of the real process. This prevents possible engine accident condition, which could have disastrous financial or environmental threats, or may endanger human life. If we implemented a hybrid Petri net into programmable logic controller [2] (PLC or on other platforms), we can use visualization to monitor active hybrid Petri nets, which is connected to a real process with input and output modules. Controlled process can be further identified and compared with the desired or expected results. Ideally should be the real verification process 100% identical, but in practice often only comes close to this value.

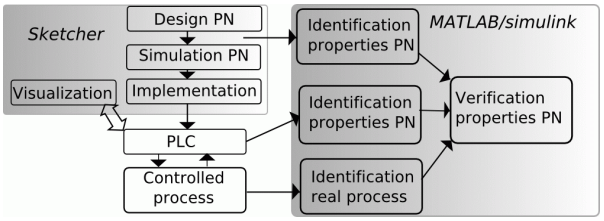


Fig. 1. Verification hybrid Petri nets

3 Conclusion

The already created tool Sketcher [3] is suitable for design, simulation, visualization and implementation of Petri Nets into a Programmable Logic Controller. In CLG laboratory is being developed PN_CLGtoolbox for MATLAB/Simulink for Hybrid Petri Nets design. The toolbox currently supports reading, writing and displaying parameters from PLC in real-time.

Acknowledgments This work has been supported in part by Ministry of Education, Youth and Sports of the Czech Republic (Research Intent MSM0021630529 Intelligent systems in automation), Grant Agency of the Czech Republic (102/09/H081 SYNERGY - Mobile Sensoric Systems and Network) and by Brno University of Technology.

References

1. David, R., Alla, H.,: Discrete, Continuous and Hybrid Petri Nets, Springer-Verlag (2005)
2. Chomát, L.: Petri net in environment of program logic controller B&R, Diploma thesis, VUT Brno (2006)
3. Chomát, L.: Paper about Petri Nets, GRAFCET, Ladder Diagram and Manual for software Sketcher, Rovira I Virgili Tarragona (2008)

Instruction Pipeline Modeling using Petri Nets

Adam Husár, Tomáš Hruška, Karel Masařík, Zdeněk Přikryl

Brno University of Technology, Faculty of Information Technology
{ihusar, hruska, masarik, iprikryl}@fit.vutbr.cz

Abstract. This paper deals with instruction pipeline modeling using Petri Nets. Such model would be useful for processor pipeline design, verification and also for instruction scheduling in C compiler backend. This paper presents ongoing work and mainly states what requirements we have on such model.

1 Introduction

Embedded systems are more and more performance demanding, especially in the areas of multimedia processing, software radio and diverse pattern recognition. One possible solution is to improve application performance by using application-specific instruction set processors (ASIPs), where instruction set extensions (ISEs) can speed the critical parts of application up. This gets us to project Lissom, where we develop an ASIP development environment. Using it, the user can describe processor architecture and microarchitecture with an architecture description language called ISAC.

Architecture description languages (ADLs) are languages targeted at processor design. Three of the most popular languages that are used in industry to develop processors are LISA from CoWare, nMl from Target and TIE from Tensilica. Our language ISAC is partially based on LISA. ADL models try to capture all the information needed for generation of toolchain (mainly compiler and assembler), simulator and hardware description.

2 Pipeline Modeling in ISAC

Two types of processor models can be designed using ISAC: 1) architectural that describe only the HW/SW interface and 2) microarchitectural, where architecture implementation using instruction pipeline is also captured.

Instruction set syntax and binary coding is in ISAC modeled using context-free grammars [4]. Instruction behavior is described in C language. Together with constructs for delayed execution, we can define the instruction pipeline. There are two ways how to describe pipelines in ISAC.

First approach is hardware-oriented. Here is each pipeline stage described using a C function that reads input latch registers and stores stage result in output latch registers.

Second approach is instruction-set-oriented. In a description of an instruction behavior, user may specify what action should be executed in specific clock cycles after the instruction was decoded.

Microarchitectural models defined in ISAC using instruction-set-oriented approach tend to be rather large (5000 lines for a model of microcontroller 8051), compared to hardware-oriented approach (2500 lines for VLIW DSP Chili 2 from OnDemand Microelectronics and 2000 lines for partial model of VLIW DSP C6400 from Texas Instruments), so even if the instruction-set-oriented approach was planned to be more abstract, the advantage is disputable. On the other hand, when using the hardware-oriented approach, every little detail of the pipeline must be described. None of these two approaches is very much suitable and we need to find a model that is as abstract as possible and reflects the pipeline behavior.

3 Approaches to Modeling Pipelines Using Petri Nets

The main reason to model pipelines using Petri Nets is that they allow to describe instruction flows, hazards, pipeline stalls and memory subsystems in a straightforward way. Also, one of the most important requirements on the model is verification. Here can be already existing Petri Nets analysis tools used.

For example in paper [1], the authors use for pipeline description so-called Reduced Colored Petri Nets, where two types of tokens are used. First type represents instructions and may carry additional information. Second type is a simple synchronization token. Another approach is described in [3]. Even though the pipeline modeling did not receive much attention yet, the problematics of hardware modeling using Petri Nets is already quite explored [2].

As stated in the abstract, this paper presents ongoing work and the exact modeling style that will be used is still under consideration. This paper accompanies a poster, where some examples of considered approaches will be presented.

This research was supported by the grants of MPO Czech Republic FT-TA3/128 and FR-TI1/038, by the Research Plan MSM No. 0021630528, by the doctoral grant GA CR 102/09/H042 and by the BUT FIT grant FIT-SS-10-1.

References

1. Reshadi, M., Dutt, N.: Generic Pipelined Processor Modelling and High Performance Cycle-Accurate Simulator Generation. In: Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, (2005) 786–791
2. Yakovlev, A. V., Koelmans, A. M.: Petri Nets and Digital Hardware Design. In: Lectures on Petri Nets II: Applications, (1998) 154-236, ISBN 978-3-540-65307-3
3. Razouk, R. R.: The Use of Petri Nets for Modeling Pipelined Processors. In: Proceedings of the 25th ACM/IEEE Design Automation Conference, (1988) 548–553
4. Lukáš, R., Hruška, T., Kolář, D., Masařík, K.: Two-Way Deterministic Translation and Its Usage in Practice. In: ISIM'05, Ostrava, CZ, 2005, p. 101–107.

BDD-based Bounded Model Checking for Elementary Net Systems*

extended abstract

Artur Męski[1], Wojciech Penczek[2,3], and Agata Półrola[1]

[1] University of Łódź, FMCS, ul. Banacha 22, 90-238 Łódź, Poland
{meski,polrola}@wmi.uni.lodz.pl
[2] Polish Academy of Sciences, ICS, ul. Ordona 21, 01-237 Warsaw, Poland
[3] University of Podlasie, ICS, Sienkiewicza 51, 08-110 Siedlce, Poland
penczek@ipipan.waw.pl

The process of design and production of both systems and software – among them the concurrent ones – involves testing whether the product conforms to its specification. To this aim, various kinds of formal methods can be applied. One of the possible approaches, widely used and intensively developed, are model checking techniques.

The main idea of model checking consists in representing a system to be verified in a form of a transition system (model), representing a specification as a temporal logic formula, and checking automatically whether the formula holds in the model. Unfortunately, the practical applicability of the approach is usually limited due to the *state explosion problem*: the state space of the system tested, which is to be searched when testing whether the formula holds, grows significantly for large concurrent systems, which follows, among others, from representing concurrency of operations by their interleavings.

One of the methods used to overcome the above problem is to apply a symbolic model checking technique. A wide class of these techniques exploits various kinds of decision diagrams to represent the model [3, 6, 7], among them reduced ordered binary decision diagrams – ROBDDs. Another branch involves SAT-based verification methods.

Bounded model checking (BMC) is an efficient verification method whose main idea consists in considering a model truncated up to a specific depth. There exist numerous papers which deal with that approach in the context of SAT-based verification for existential temporal properties: a model checking problem on a fraction of the model is translated into a test of propositional satisfiability, which is then performed using a SAT-solver [1, 9, 10]. SAT-based BMC for elementary net systems and the existential fragment of the Computation Tree Logic (CTL) was considered in [8].

Performing bounded model checking using BDDs instead of SAT was investigated in [4, 2]. However, both these papers focused on a limited class of properties. On the other hand, Jones and Lomuscio [5] presented a BDD-based

* Partly supported by the Polish Ministry of Science and Higher Education under the grant No. N N206 258035.

bounded model checking for the existential fragment of the epistemic logic CTLK and interpreted systems.

The aim of our work is to present a BDD-based BMC verification for elementary net systems and the existential fragment of the logic CTL. Although the novelty of the approach is not significant, we expect to obtain a technique of a better efficiency than the previous SAT-based one shown in [8]. Such an expectation follows from the encouraging results of Jones and Lomuscio presented in [5]. We are going to provide a comparison of the experimental results.

In order to obtain a possibly complete verification tool for elementary net systems, we are also going to describe and implement a BDD-based BMC verification of LTL properties.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. of the 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.
2. G. Cabodi, P. Camurati, and S. Quer. Can BDD compete with SAT solvers on bounded model checking? In *Proc. of the 39th Design Automation Conference (DAC'02)*, pages 117–122, 2002.
3. G. Cabodi, S. Nocco, and S. Quer. Mixing forward and backward traversals in guided-prioritized BDD-based verification. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 471–484. Springer-Verlag, 2003.
4. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 436–453. Springer-Verlag, 2001.
5. A. Jones and A. Lomuscio. A BDD-based BMC approach for the verification of multi-agent systems. In L. Czaja, editor, *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'09)*, volume 1, pages 253–264. Warsaw University, 2009.
6. A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of the 20th Int. Conf. on Applications and Theory of Petri Nets (ICATPN'99)*, volume 1639 of *LNCS*, pages 6–25. Springer-Verlag, 1999.
7. J. Møller, J. Lichtenberg, H. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proc. of the 13th Int. Workshop Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 111–125. Springer-Verlag, 1999.
8. W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.
9. W. Penczek, B. Woźna, and A. Zbrzezny. Towards bounded model checking for the universal fragment of TCTL. In *Proc. of the 7th Int. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*, pages 265–288. Springer-Verlag, 2002.
10. O. Strichman. Tuning SAT checkers for bounded model checking. In *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 480–494. Springer-Verlag, 2000.